Eclipse-Based Front-End for OMS^{jp}

Diploma Thesis

Barbara Aeppli

<barbara@computerscience.ch>

Prof. Moira C. Norrie
Michael Grossniklaus

Global Information Systems Group
Department of Computer Science
ETH Zurich

February 17th 2005

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

globis

Abstract

OMS^{jp} is a Java interface which makes access to heterogenous OMS platforms possible in a uniform manner. To enhance software development based on **OMS**^{jp}, the aim of this diploma thesis was to build a graphical user interface on top of **OMS**^{jp} for modelling and administrating OMS databases. The resulting front-end has the advantage that it is applicable to all OMS platforms supported by **OMS**^{jp}. To facilitate the implementation process, the front-end has been implemented as a plug-in for the widely-used Eclipse Platform. Eclipse offers an extensible architecture and supporting functionality common to front-end applications. Furthermore, the popularity of Eclipse leads to a reduced learning effort for the new OMS front-end since it adopts well-known Eclipse paradigms and conventions.

Contents

1 Introduction

During the last couple of years the Global Information Systems Group of the ETH Zurich has developed an object-oriented database management tool suite, called OMS [8]. OMS consists of a series of tools and technologies designed to support database development. At the moment most of the tools that belong to the OMS suite offer their own graphical user interface (GUI) or so called front-end. With the introduction of **OMS**^{jp} [9], a Java library providing access to OMS databases, it becomes possible to develop a single front-end that can be used uniformly for all OMS platforms.

1.1 OMS Suite

The OMS suite is based on its own object model, OM [6]. OM is a generic object data model which specifies constructs and operations for the management of interrelated collections of data objects. It is designed to support database development from conceptual design through to implementation independent of a particular programming language environment.
Up to the present the OMS suite comprises a number of platforms which implement the OM model using different programming environments such as Prolog, Java, Oberon or Python. The platform implemented in SICStus Prolog [10] is called OMS Pro [7]. OMS Pro supports rapid prototyping, but it can be used for the implementation phase as well. OMS Pro provides a TCL/Tk [12] graphical user interface for browsing, querying and manipulating database objects. The user interface is rather advanced but it still has some deficits. Another platform that belongs to the OMS family is eOMS. eOMS is based on PostgreSQL and implemented in Python. It offers a concurrency control mechanism to allow multi-user access.
These different OMS platforms are very heterogenous and do not provide a common Application Programming Interface (API). OMS Pro offers a collection of Prolog predicates as API and hence no interface that can be accessed directly from Java. This is not very convenient for application developers who want to access OMS databases from a Java application.

Therefore **OMS**^{jp} was developed and became the latest member of the OMS suite.
OMS^{jp} is a uniform Java library that allows transparent access to heterogenous OMS platforms as back ends. **OMS**^{jp} maps the OM model of the database into concepts that can be used in a Java application. An application built using **OMS**^{jp} can run with various OMS platforms such as OMS Pro and eOMS if an adequate driver for the underlying platform is available (see [4]). At the moment this is only the case for OMS Pro, but support for eOMS is planned. Figure 1.1 shows an overview of the **OMS**^{jp} architecture.

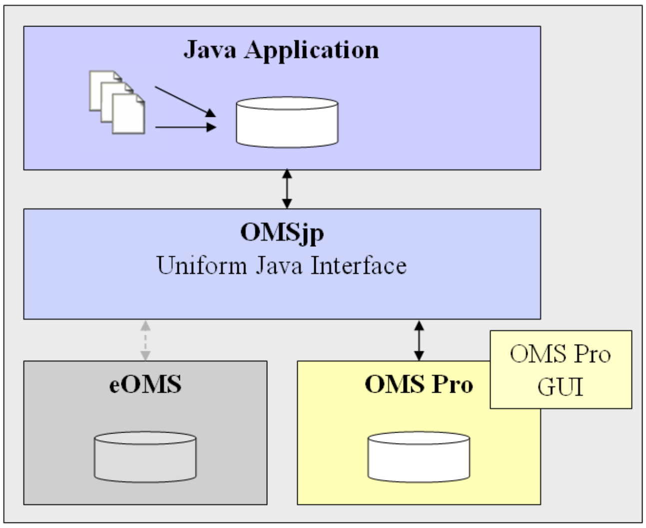

Figure 1.1: Architecture Overview

The fact that database supported applications using **OMS**^{jp} are flexible to freely exchange their underlying OMS platform inspired the idea of implementing a single OMS front-end based on **OMS**^{jp}. Such a front-end is applicable to all OMS platforms supported by **OMS**^{jp}. This implies that only one graphical user interface has to be implemented and maintained for several platforms which leads to a reduced development effort.
Since **OMS**^{jp} is a Java interface it was not under consideration that a front-end on top of **OMS**^{jp} should be implemented using a different programming language than Java. Otherwise there were no further constraints what technique to use. A stand-alone Java Swing application would have been possible, but it was decided to integrate the front-end into the Eclipse [2] environment.

1.2 Motivation for Using Eclipse

Eclipse is an Integrated Development Environment (IDE) that can be used for developing applications in various programming languages. Most commonly it is used for developing Java applications and during the last few years the number of developers that are using Eclipse increased considerably. One of the reasons for this trend might be that the user is able to

customise his development environment suitable to his needs.
Eclipse is an open source project offering an extensible architecture. This fact enables developers to integrate their own specialised tools seamlessly into the Eclipse environment. In addition to the extensibility, Eclipse offers a lot of functionality and powerful tools that simplify the implementation of graphical user interfaces. This is very convenient from a developer's point of view and a valuable basis for the integrating of an OMS front-end into Eclipse.
The mechanism of integrating one's own tools into Eclipse gives the users the possibility of composing a development environment from a set of highly specialised tools. Users are able to choose which tools they would like to use and can therefore only install the tools they need to perform their tasks. This fact helps to keep Eclipse manageable and reduces the risk of having a completely overloaded user interface.
When it comes to developing Java applications that are built on top of an OMS database, a further advantage of having an OMS front-end tool directly available within Eclipse emerges. The developer can use one single tool for the implementation process of the Java application as well as for the modeling and administration of the underlying OMS database. This eliminates the switching between applications and makes working more comfortable. Beginners that implement a Java project coupled to an OMS database for the first time only need to get acquainted with one development environment. Or even better, if they already know how to work with the Java development tools provided by Eclipse, it is only a small step to learn how to work with OMS databases since the OMS front-end integrates into the Eclipse user interface and adopts its paradigms and conventions.

1.3 Document Structure

After getting acquainted with the Eclipse environment the first task of the diploma thesis was to evaluate the existing OMS Pro graphical user interface. The goal of the evaluation was to find requirements for the new front-end. The findings of this evaluation and the derived requirements are presented in chapter 2. Chapter 3 gives a short overview of Eclipse. The main concepts concerning the extensible architecture of Eclipse and its user interface paradigms and conventions are introduced. The next step was to work out some design proposals which were then discussed to decide on a final design for the OMS front-end. The difficulties were to find suitable solutions that improve the weaknesses of the existing OMS Pro GUI and respect the conventions of Eclipse at the same time. The results of this design phase are covered in chapter 4. The design phase was followed by the actual implementation phase. Chapter 5 presents technical details about the implementation whereas chapter 6 summarises the main conclusions.

2 Evaluation of the OMS Pro Graphical User Interface

An evaluation of the existing OMS Pro graphical user interface (GUI) served as a starting point for the design of the OMS Eclipse plug-in. The goal of the evaluation was to explore the strengths and the weaknesses of the existing OMS Pro GUI. Concepts that were considered as especially useful should be adopted for the new Eclipse plug-in whereas weaknesses should be enhanced in order to improve the usability of the application considerably.
This chapter provides the main findings of the evaluation and it summarises the requirements for the eclipse-based front-end that were derived from the findings.

2.1 Methodology

The evaluation of the OMS Pro GUI as it was carried out for this diploma thesis was not a scientific study in the sense that a lot of different persons were interviewed according to predefined questionnaires which were then analysed and interpreted. But the evaluation is based on personal experiences we made while working with OMS Pro during the last couple of years and on discussions we had with other people that are using the system as well.

2.2 OMS Pro Concepts and Terminology

The OMS Pro GUI is implemented using TCL/Tk and consists of a main application window and a few additional windows that can be opened from the main window through user interaction.
The main application window is shown initially. It can be seen in figure 2.1. The following list describes the most important elements of the main application window.

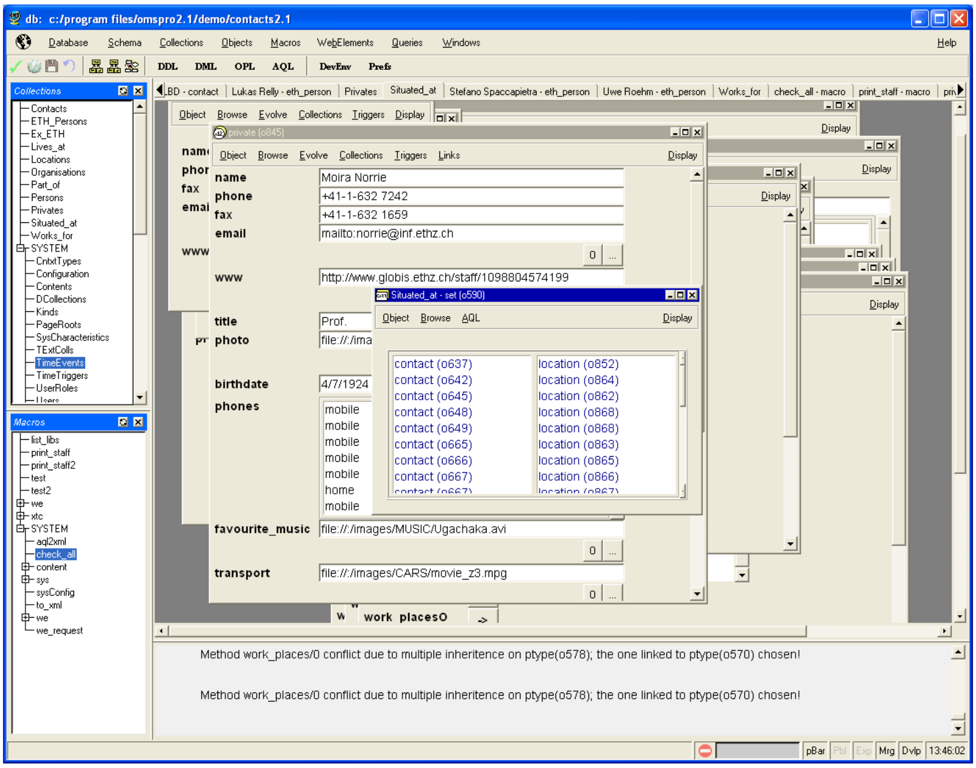

Figure 2.1: OMS Pro Main Application Window

- Menu Bar
  The main functionality can be found in the menu bar. Context menus using the right mouse button are not available in the OMS Pro main window.

- Tool Bar
  Some important actions as for example commit and rollback actions are represented by a short cut button in the tool bar.

- Collection Panel
  User as well as system collections are placed in a panel that is visible all the time. It is also possible to open collections using the ‘Collection’ menu in the menu bar, but the panel allows to have these objects clearly arranged and accessible with fewer mouse clicks.

- Macro Panel
  The same concept as for collections exists for macros. The panel displays all user and system macros, the latter as an optionally expandable tree node.

- Main Panel / Object Area
  All objects that the user opens are displayed somewhere within the main panel or object area of the application. Each object is represented as a rectangular object window and additionally with a tab in the tab bar. The tab bar is at the top of the main panel.

- Console
  The console window provides system and error messages to the user.

Graphical schema editors can be opened via the menu or tool bar in the main window, but they are displayed in separate windows. The system offers a base type and an object type editor (see figure 2.2) to add, edit and remove new types and a classification editor (see figure 2.3) to add, edit and remove collections, associations and constraints. These schema editors offer context menus in addition to the tool and menu bar.

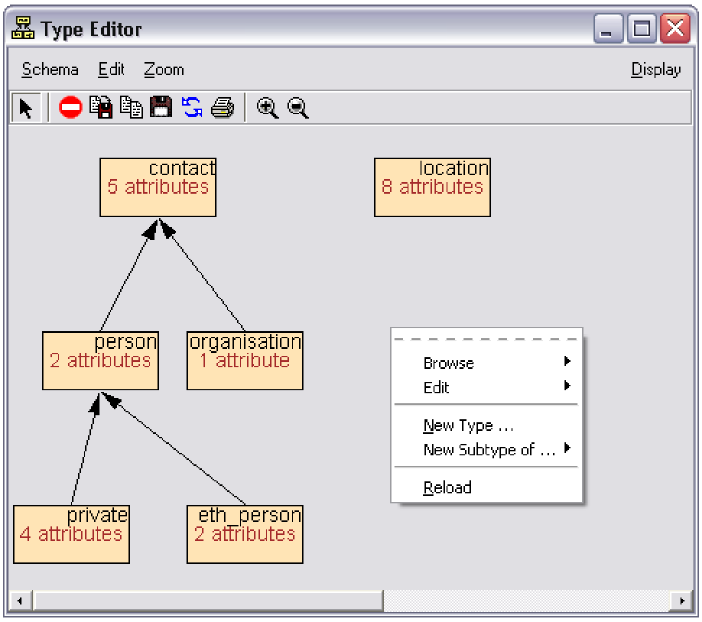

Figure 2.2: OMS Pro Object Type Editor

2.3 Evaluation Results

The following two sections provide an enumeration of the most important strengths and weaknesses of the OMS Pro GUI.

2.3.1 Strengths of the OMS Pro GUI

Storing Paradigm OMS Pro applies a commit/rollback paradigm as storing paradigm which is very convenient for the user. All changes on open database objects affect the state of the database immediately. This allows the user to work efficiently without having to perform 'Save' operations every time a change has been made. Between two 'Commit' operations the database can be in an inconsistent state without getting any error messages. The 'Rollback' operation allows to discard all changes since the database was last committed.

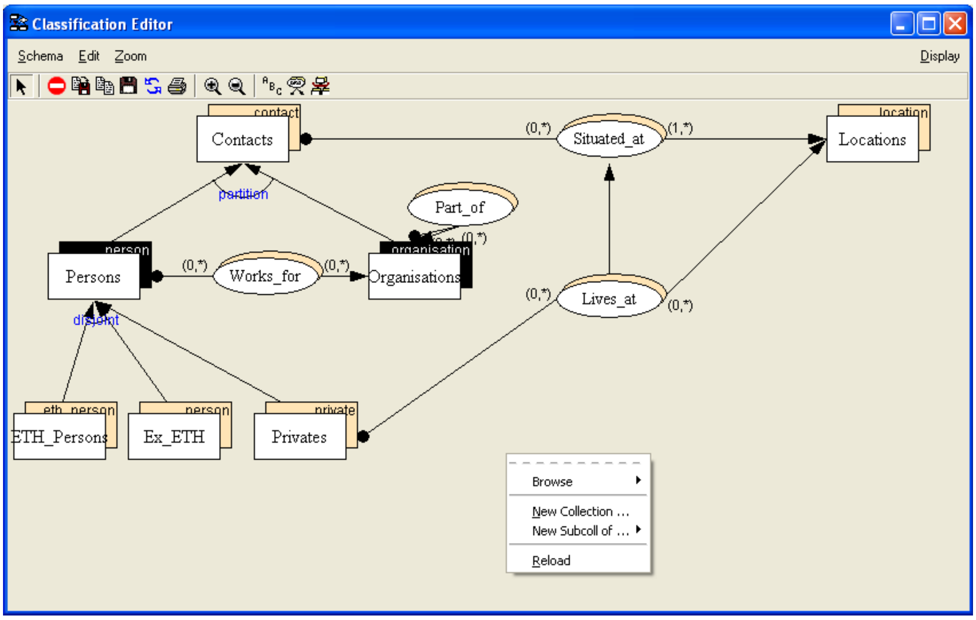

Figure 2.3: OMS Pro Classification Editor

Context Information In OMS Pro every object that is displayed in the object area provides a menu bar. Depending on the kind of object, the menu bar offers different menu items that deliver information about the object to the user. The menu item ‘Links’ shows all objects that are associated to this object. This is a very useful concept since the user can immediately understand the context of an object. The ‘Collections’ menu shows to which collections the object belongs and to which collections it could be added. All possible types that an object can have are listed in the ‘Evolve’ menu.

Collection and Macro Panel The collection and the macro panels are convenient for the user since they allow the user to gain an overview of the database and fast access to frequently used objects.

Graphical Schema Editors The graphical schema editors are helpful tools since they visualise relations between collection and type objects. In the base and object type editor super- and subtype relations are shown whereas in the classification editor super- and subcollection relations, constraints and associations between collections are represented graphically. Having such a graphical representation gives the user a valuable overview over the database. Further it facilitates schema changes and the browsing of these objects since they can be opened directly from the schema editor by double-clicking.

2.3.2 Weaknesses of the OMS Pro GUI

The weaknesses that were found are categorised into three major categories: graphical, model and usability problems.

Graphical Problems

This category comprises problems that arise due to a poor graphical representation of components or due to an inaccurate implementation. A more careful implementation or an implementation using a different graphic library might solve these problems.

Window Focus Dialog windows as for example the preference window or the graphical schema editor windows tend to loose their focus unprovoked. They often disappear in the background behind the main application window despite the fact that they were still active. The user either thinks that the window was closed and chooses the corresponding menu entry again to reopen the window or he has to minimise the main application window to be able to reset the focus manually.

Object Window Size Database objects are opened in the main panel of the main application window. They are displayed as small object windows. The size of these windows is in many cases unadapted. Buttons that are at the bottom of the window to cancel or execute an action are not visible at once. The user first has to resize the object window in order to make these buttons accessible. If the object window contains child components as for example a scrollable list, these components do not resize if the window is resized. If such a list contains more entries than it can display, the user still has to scroll in the list even though the window may be big enough to show all list entries. Furthermore the maximisation function of objects windows in not useful at all. The maximised object windows become so big that the user is forced to scroll in order to make the 'close' button visible again.

Arrangement of Object Windows The arrangement of the object windows in not satisfactory. Depending on the number and positioning of already open object windows in the main panel, it can happen that a new object is opened in a way that only a part of the object is actually visible. The user first of all has to scroll the main panel or to displace the object in order to centre the object in the main panel and actually look at the whole object content.

Model Problems

Model problems are problems where the result of a function or action does not correspond to the mental model of the user. A mental model is a set of beliefs that a user holds about how things or in this case software features work. Such a mental model is often established based on former experiences with tasks that are similar to the one that should be currently solved. If a user is solving a task and the result of an action is different from his expectations, the user is confused and the learning process is slowed down. The user cannot revert to something known and even worse he has to react differently than in another known case.

File Browsers The file browsers offered in OMS Pro are not intuitive to use. They do not correspond to the expectations users have from working with Windows or Linux. The file structure is not displayed as a tree but as a list. To navigate between hierarchy levels the user can either go up or down one hierarchy level at a time or he can type the target directory manually in the selection field. If the nesting of directories is very deep, the navigation in the file system is rather tedious. Typing the target directory manually assumes that the user knows the exact path by heart and is able to spell it correctly.
In addition to the exhausting navigation, the hierarchy levels are not strictly separated. If the file browser shows drive letters, the selection of a drive results in a list of drive letters followed by the directories of the chosen drive. For the user it is not obvious to which drive the listed directories belong.
The file browsers do not provide a state memory. They are not able to store the location in the file system the user accessed last.

Tab Bar Every object that is opened in the object area of the main application window is represented by a tab in the tab bar. The tab bar is displayed at the top of the object area. The problem with the tab bar is that the ordering of the tabs does not correspond to the opening order of the objects. This is rather irritating for the user and makes it difficult to retrieve already open objects. The user expects to find the tab of the last opened object at the very right of all open tabs. Furthermore humans rely on a visual orientation and learn which object corresponds to which tab without actually reading the tab label. OMS Pro does not follow these principles. It does first of all a grouping of tabs according to the object type and within objects of the same type the tabs are ordered alphabetically. One consequence is that every time an object is opened the tab ordering changes and the user has to repeat reading tab labels carefully to retrieve an object.

Adding Associations The process of adding an association between two objects is rather circumstantial. The user has different possibilities to establish a link between objects.
The most natural approach is to add a link starting from the source object and then indicating the target object. OMS Pro does support this approach offering the 'Links' menu that can be found in the menu bar of every object window. The 'Links' menu provides a list of all possible associations the object could be part of and all existing links to other objects. The user can then add a new link using the menu entry 'New' in the 'Links' menu. The source object of the new link is already clear, but OMS Pro displays a dialog window where the user has to specify the source as well as the target object. The value fields are preselected but the preselection does not correspond to the object that should be linked. To set the value fields the user can directly enter the object id which he has to look up or to know by heart. An auto complete function is available but it does only work when the preselection is cleared. Another way to set the value field is to use 'object fishing' which means that the user has to click on the object to fill in. This requires that the object to click on is opened in the object area. If the user finally sets the two objects that should be linked, the user can choose between the operations 'Insert', 'Replace' or 'Remove'. This is not consistent with the fact that the dialog window appeared after clicking on a menu entry called 'New' which has nothing to do with replacing or removing links.

Another approach to add an association is to open the binary collection that is based on this association and to add a new entry to this collection using the same dialog window as in the previous approach.

Usability Problems

Usability problems reduce the usability of the system and make it rather tedious to work with the system. Solving these problems should lead to a more user-friendly system.

Overlapping Windows in Object Area OMS Pro allows to open numerous object windows in the object area of the application window. These object windows overlap and with an increasing number of open objects the object area tends to become more and more crowded. It gets difficult to work with these objects since there exists no clear arrangement and the growing number of open objects produces a rather disturbing impression on the screen.

Application State Memory OMS Pro does not provide an application state memory. Such a memory mechanism is responsible for saving and restoring the state of a dialog, an object or of the application. If the OMS Pro application is closed, all open objects are closed as well. If the user restarts the application he has to reopen all objects on his own. If some objects are used quite frequently or if the user was in the process of modifying an object (for example writing a macro) it might be helpful if the system could remember which objects were open.
Furthermore the collection and macro panel do not have a state memory. If the user expands the system collections in the collection panel and then adds a new collection to the system, the system collections collapse and have to be expanded again.

Window Switches Editing a database in OMS Pro requires many window switches between the main application window and the graphical schema editor windows since some tasks are easier to accomplish in one or the other window. An example is the task of adding a new association between two collections. The process of adding the association is easy to do in the schema editor by connecting the two corresponding collections with a line. But if the user wants to edit the cardinalities of the association he has to switch back to the main window and edit the cardinalities in the association object. These switches make it difficult to keep an overview, especially since the graphical schema editor windows are independent windows which are not integrated in the main application window.

Data Separation The user and system data are not clearly enough separated. It might be the case that the user would like to have a list with all methods he added to the database. Methods are not listed in the associated type but a possibility to get a list is to user the ‘Match’ functionality in the ‘Objects’ menu. The result of this is a list with user and system internal methods. This makes it rather difficult to find all user defined methods. Having access to the system data is an advantage but in the user interface a clear distinction of user and system data is preferred.

Hidden Functionality The creation of a new object assumes an already defined object type. The functionality to create an object is visible at first sight in the menu bar whereas the functionality to create a type is hidden and not obvious. The creation of types can be done in the graphical object type editor or using DDL. The same is true for adding associations, collections or methods.

Object IDs Most of the time the user has to work with object ids which is not very natural for humans. Object ids are excellent for the system to manage objects but they are quite abstract for the user. For the user it is a lot easier to remember a name than an object id. OMS Pro makes some attempts to use attribute values together with object ids to facilitate working with objects. Examples are the tab bar or when opening a collection through the collection panel. This technique should be introduced consistently throughout the whole application.

2.4 Requirements

The aim of the evaluation was to establish requirements for the OMS Eclipse plug-in. The evaluation resulted in a number of positive and negative aspects described in section 2.3 which form a valuable basis to derive requirements. The idea was to adopt the strengths of OMS Pro as far as possible and to improve the weaknesses at the same time.
The following list summarises the main requirements for the OMS Eclipse plug-in.

- Keep Storing Paradigm
  The commit / rollback paradigm that OMS Pro uses is very convenient for the user since the user does not have to care about numerous save operations and no changes are lost due to a forgotten save operation. Therefore the OMS Eclipse plug-in should adopt the storing paradigm if possible.

- Offer Context Information
  One of the strengths of OMS Pro is the context information that is provided for a database object. The user is able to understand quite quickly how an object is related to other objects and how it is integrated in the database. A requirement for the OMS Eclipse plug-in is thus to make context information easily accessible.

- Offer Graphical Schema Editors
  The graphical schema editors are an essential element of the OMS Pro graphical user interface since they help the user to obtain an overview of the database really fast. Such graphical schema editors should also be part of the OMS Eclipse plug-in. But it is also important that all tasks can also be executed without graphical schema editors to reduce the window switching on the one hand. On the other hand functionality that is only available in the graphical schema editors is not obvious to find and should therefore also be accessible directly in the main window.

- Extend Object Panels
  OMS Pro offers collection and macro panels. They are extremely helpful to gain an overview over the existing database objects and to access them quickly. Therefore the

OMS Eclipse plug-in should offer such panels as well and in addition to collection and macro panels, similar panels for other database objects such as types or methods should be considered.

- Redesign Object Arrangement
  The object arrangement in the object area is not optimal. If a number of objects are open it gets quite hard to work with these objects since the object area tends to become very crowded and objects can only be found through the tab bar. Attempts should be made to propose a different approach for the object area.

- Provide Intuitive Tab Ordering
  Having an intuitive tab bar is important since tab bars are commonly used. The OMS Eclipse plug-in should thus introduce a tab bar that allows to retrieve objects efficiently.

- Reduce Use of Object IDs
  Since object ids are not a natural way of working with objects for humans, they should be extended or replaced through meaningful attribute values.

- Enhance Process of Adding Associations
  The process of adding associations between two objects is rather circumstantial. This process should be simplified by offering more intuitive ways of introducing links.

- Introduce Application State Memory
  An application state memory is useful since it saves time while working with a system. If a user opens a file in a certain directory the chances are high that the next file that is accessed is located in the same directory or nearby. Keeping track of user preferences and choices is convenient for the user.
  Another case where it makes sense to store the state is when closing and reopening the system. Often the user resumes his work where he finished and it is helpful for the user if the system has exactly the same state as before the application was closed. The introduction of an application state memory is therefore a requirement for the OMS Eclipse plug-in.

3 Eclipse Platform

3.1 Introduction

Eclipse is a universal platform for integrating development tools into the Eclipse workbench. The Eclipse Platform is implemented in Java and it is simply a framework and a set of services for building a development environment from plug-in components. Eclipse has no specific knowledge about any domain, and is not limited to Java language applications. However, the Eclipse SDK includes the Eclipse Platform and a set of plug-ins including the Java Development Tools (JDT) and the Plug-in Development Environment (PDE). The Eclipse Platform, the JDT and PDE plug-in form together the Eclipse project as it is shown in figure 3.1.
Eclipse is an open source project and together with its extensible architecture, it allows software developers to extend Eclipse and to build tools that integrate seamlessly with the Eclipse environment. Since Eclipse 3.0 the developer can also choose to build stand-alone rich client applications that are simply using the platform runtime and its plug-in mechanism. With this Rich Client Platform (RCP) approach the spectrum of possible applications grows enormously since the user interface and the underlying model of the application do not have to integrate into the Eclipse workbench and workspace anymore. This fact enables the developer to implement almost freely designed applications.
Section 3.2 explains the components of the Eclipse Platform. In section 3.3 a survey of the Eclipse plug-in architecture can be found and finally section 3.4 provides information about the Eclipse graphical user interface and its conventions.

3.2 Platform Overview

Eclipse Platform is the name for the core framework and services upon which plug-in extensions are created. It provides the runtime in which plug-ins are loaded and run. The Eclipse

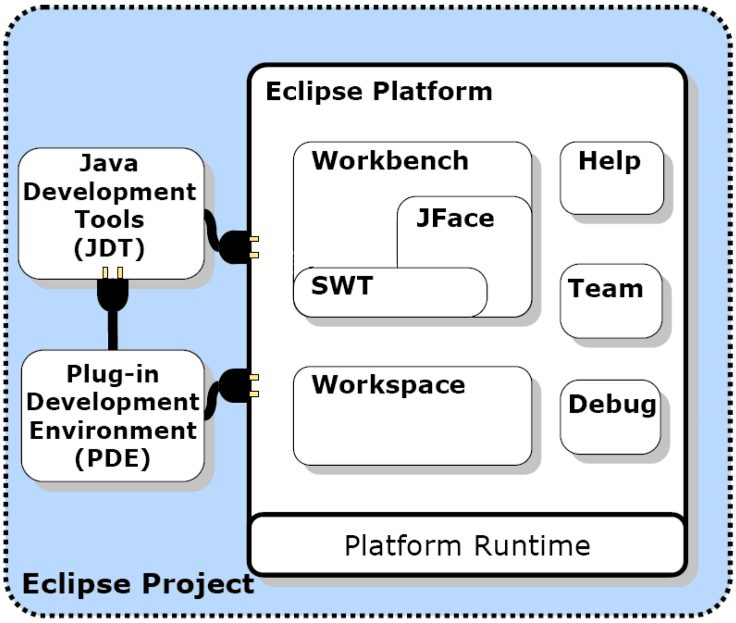

Figure 3.1: Eclipse Project Overview

Platform itself is not really a true product that would ship by itself. Its direct consumers are tool builders since they add the value to the Eclipse Platform that makes it useful to end-users. The Eclipse Platform is basically just the nucleus around which tool builders build tool plug-ins.

The Eclipse Platform is divided up into the core part and the user interface (UI) part. Anything classified as user interface needs a window system, whereas components that belong to the core can run without window system. The user interface part of the Eclipse Platform is known as the Workbench. The core part is simply called the Platform Core.

Figure 3.2 shows the major components of the Eclipse Platform.

Platform Runtime The platform runtime is a micro-kernel that is responsible for discovering, integrating, and running plug-ins. At start-up the runtime environment discovers what plug-ins are installed and creates a registry of information about them. To reduce start-up time and resource usage, it does not load and activate any plug-in until the plug-in is actually needed. This mechanism is called lazy loading and it is used consistently throughout Eclipse. Except for this runtime kernel, all functionality in Eclipse is implemented as a plug-in.

Workspace The workspace is the plug-in that is responsible for managing the user's resources. Resource is the collective term for projects, folders and files. The workspace consists of one or more top-level projects whereas each project maps to a corresponding user-specified directory in the file system. Each project contains folders and files that are created and manipulated by the user. All folders and files in the workspace are directly accessible to the

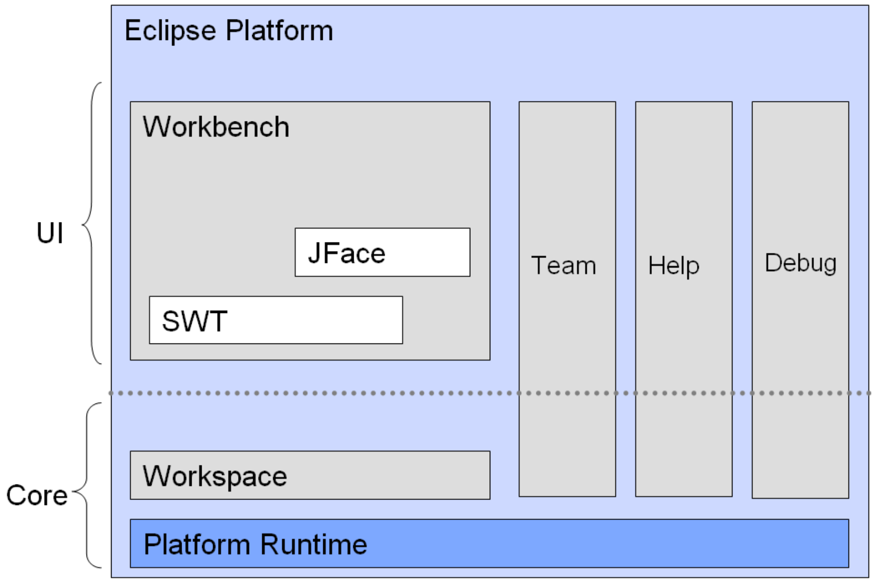

Figure 3.2: Eclipse Platform

standard programs and tools of the underlying operating system. Tools integrated in the platform are provided with an API for dealing with workspace resources. The workspace is also responsible for notifying other interested plug-ins about resource changes, such as files that are created, deleted or changed.
In earlier versions of Eclipse tools were based on files and folders and therefore tightly coupled with workspace resources. Recently the Eclipse developers got rid of this coupling and made the workspace and resources an optional part of the platform. It is now possible to have configurations of the platform that will not have a workspace. This allows tools to work with other data sources such as database objects or data streams.

Workbench The workbench provides Eclipse with an extensible user interface. The workbench API and its implementation are built form two tool kits:

- SWT - Standard Widget Toolkit
  The Standard Widget Toolkit is a graphical user interface toolkit for the Java programming language. It was created by the developers of the Eclipse project to provide access to the native user interface facilities of the operating systems that host Eclipse. The SWT graphics library offers a common OS-independent API for widgets and graphics, but it is more closely mapped to the native graphics capabilities of the underlying operating system than Swing or AWT. This fact not only makes SWT faster, but also allows Java programs to have a look and feel more like native applications. The Eclipse Platform and most of the tools that plug in to it use SWT for presenting information to the user.

- JFace - a UI toolkit built on top of SWT
  JFace is a higher level user interface construction toolkit that is designed to be used in conjunction with SWT. JFace provides classes for handling and simplifying common GUI programming tasks and its API as well as the implementation are window-system-independent. It includes standard GUI toolkit components such as dialogs, wizard frameworks and progress reporting for long running operations. Two additional features that are useful are actions and viewers.

Team The team component is responsible for providing support for version control and configuration management. It adds views to allow the user to interact with whatever version control system.

Help In the same way that plug-ins add functionality to Eclipse, the help component of the Eclipse platform offers a navigation structure that allows tools to define and contribute its own documentation. Raw content is added as HTML files whereas the arranging of the raw content into online guides with a suitable navigation structure is expressed separately in XML files.

Debug The debug component offers a generic debug mechanism that cannot only be used for the Java language. It includes launch configurations to specify how to run a program, a generic debug model offering breakpoints, standard debug events and actions and a generic debug user interface with its own perspective.

3.3 Eclipse Plug-In Architecture

As described in section 3.2 the Eclipse platform consists of the platform runtime which is a small kernel and a varying set of plug-ins.

3.3.1 Plug-In Description

A plug-in is the smallest unit of an Eclipse function that can be developed and delivered separately. Small tools as for example an action to create zip files are usually written as a single plug-in whereas complex tools such as an HTML editor may span several plug-ins. These plug-ins are then grouped into installable pieces, so called features.
Eclipse plug-ins are actually components that provide a certain type of service within the Eclipse environment. A plug-in is represented by an instance of a plug-in runtime class or plug-in class for short. The plug-in class provides methods that are called for the activation or deactivation process of the plug-in. All plug-in classes extend the **org.eclipse.core.runtime.Plugin** class which is an abstract class providing generic functionality for managing and configuring plug-ins.
In addition to the plug-in class every plug-in is described by an XML plug-in manifest file. This manifest file contains information about the plug-in and it tells the Eclipse runtime how the plug-in can be activated. Listing 3.1 shows a minimal plug-in manifest file.

```
<?xml version="1.0" encoding ="UTF-8"?>
<plugin>
  name = "HTML Editor"
  id = "ch.ethz.globis.test.htmleditor"
  version = "1.0.2"
  provider = "ETH Zurich"
  <runtime>
     <library name = "htmleditor.jar">
        <export name = "*"/>
     </library>
  </runtime>
</plugin>
</xml>
```

Listing 3.1: Minimal Plug-In Manifest File

Every plug-in has a unique identifier (XML attribute id) which is used to refer to the plug-in within the manifest files of other related plug-ins. Other attributes like name, version or provider just contain information about the plug-in itself. The library name attribute specifies the name of the jar library that is generated when the plug-in is deployed.

3.3.2 Plug-In Deployment and Activation

To deploy a plug-in all resources that belong to this plug-in are copied into an individual folder residing in the 'plugins' directory of the Eclipse installation directory. These resources comprise first of all the plug-in manifest file which is essential for the activation process, then all jar files containing the compiled class files and other resources such as for example an icons folder with icons used in the plug-in.
After restarting the Eclipse workbench the installation and deployment process is completed and the plug-in can be activated and used.
The activation of a plug-in involves loading the plug-in resources into memory, instantiating the plug-in runtime class and initialising its instance. Since an Eclipse environment can have an unlimited number of installed plug-ins, the available memory and a reasonable start-up time make it impossible to activate all plug-ins at system startup. Therefore Eclipse follows the principle of lazy loading which means that a plug-in is only loaded into memory if Eclipse is required to perform some function of the plug-in.
To be able to load required plug-ins, the Eclipse runtime must know all plug-ins that are currently installed. Eclipse gathers this knowledge using a discovery mechanism at system startup. This mechanism involves the parsing of all plug-in XML manifest files and these parsed plug-in specifications are then cached in an in-memory repository called the plug-in registry. If a plug-in is activated the Eclipse runtime queries the plug-in registry to obtain the plug-in runtime class to instantiate and to know what resources to load into memory. Each plug-in gets its own Java class loader.
Most of the plug-ins provide buttons or additional menu entries to the Eclipse workbench. The graphical appearance of these workbench components is also specified in the plug-in

XML manifest which allows the workbench to display these buttons and menu entries without actually loading the plug-in. Only when the user performs these actions, the plug-in is activated and the corresponding action class is called.
Plug-ins may be dependent on functions of other plug-ins. A dependency of another plug-in can be indicated in the XML manifest using the ‘requires’ tag. If a plug-in which depends on other non-active plug-ins is activated these plug-ins are loaded transitively in order to guarantee an error-free execution of the requested plug-in.

3.3.3 Extension Mechanism

As described in section 3.3.2 plug-ins can depend on other plug-ins if they use the functionality offered by the plug-in. A second kind of relationship between plug-ins are extensions. The extension mechanism is the essential mechanism used to allow plug-ins to extend the functionality of other plug-ins. For example if a plug-in which implements an HTLM editor is added to the workbench, the plug-in has to extend the general Eclipse editor plug-in in order to allow an integration of the HTML editor into the Eclipse user interface. Figure 3.3 shows schematically how the extension mechanism works.

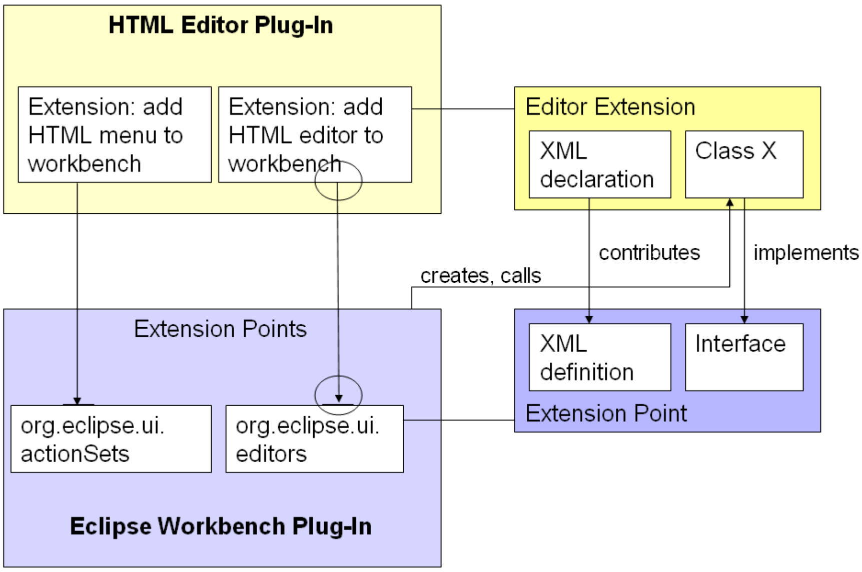

Figure 3.3: Eclipse Extension Mechanism

Any plug-in may allow any number of other plug-ins to extend it by adding processing elements which causes the plug-in to modify its behavior. In order to control these extensions made by other plug-ins, every plug-in defines precisely specified extension points. Extension points are specified using an XML configuration element and they are some kind of slots that extensions can plug into. For example the Eclipse workbench offers an extension point which allows other plug-ins to add menus to the main menu bar at a specific location.

Every plug-in can define multiple extension points and every plug-in can extend multiple extension points offered by other plug-ins.

3.4 Eclipse User Interface and its Paradigms

The workbench plug-in supplies structures in which tools interact with the user and it is therefore synonymous with the Eclipse Platform graphical user interface and with the main window the user sees when the platform is running. The Eclipse user interface has its own paradigm and follows certain conventions. Developers that contribute to the Eclipse workbench are asked to stick to the available guidelines [3].
The Eclipse platform user interface paradigm is centred around editors, views and perspectives. The user can choose which perspective to display and the chosen perspective then determines the actual appearance of the workbench. Figure 3.4 exhibits the most important workbench concepts and terminology.

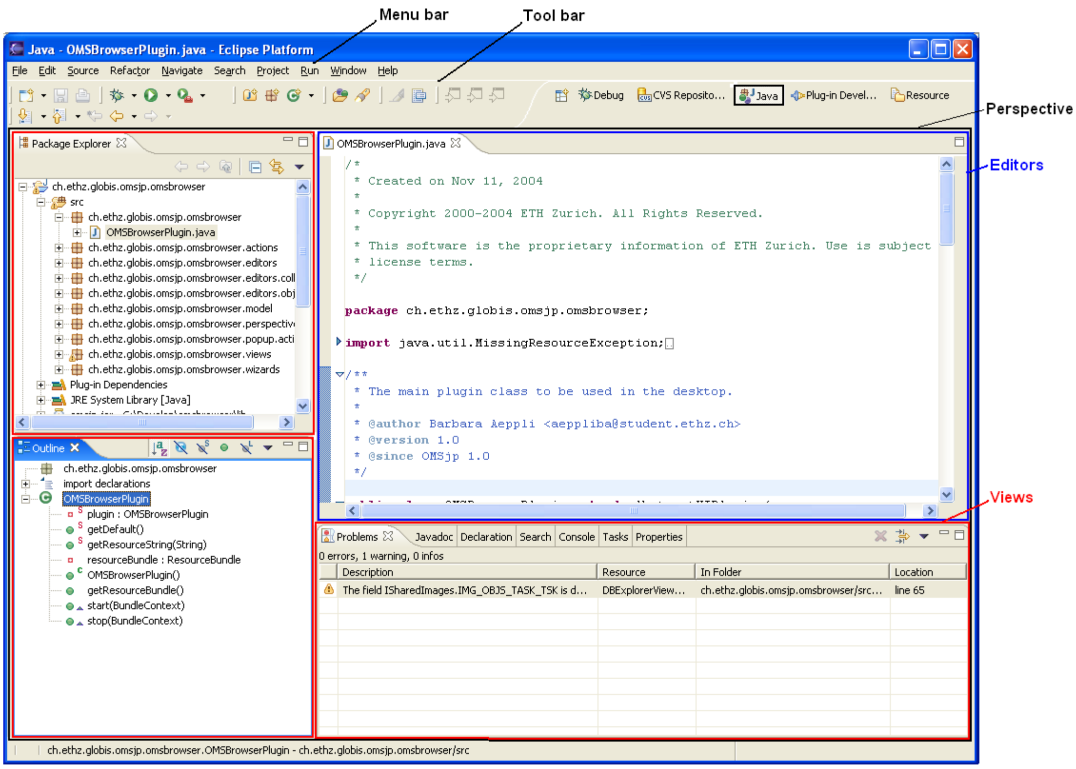

Figure 3.4: Eclipse Workbench, Java Perspective

Perspective A workbench window can have several perspectives but only one of them is visible at a time. The user chooses which perspective to display and he can quickly switch

between different perspectives if needed.
A perspective is an arrangement of editors and views and every perspective contributes special actions to the menu and tool bar. Each perspective is designed to perform a specific task, such as writing or debugging a Java program, and each of the views in the perspective is chosen to allow the user to deal with different aspects of that task. A perspective always contains a single editor area which can be hidden or not. Additionally, the perspective controls the initial view visibility, the layout how the editor area and the views are arranged and which actions are visible. The concept of perspectives helps to filter information and as a result of this, the Eclipse system scales to a large number of installed tools without having an overcrowded user interface.
Once a perspective is open, the user is free to customize it to suit his needs. The user can close views, add views from other perspectives or change the layout. As long as a perspective stays open, the customization is preserved, also across multiple sessions. More information about perspectives can be found in [1], chapter 10 or [11].

Editors Editors are opened in the single editor area of a perspective. If more than one editor is open at a given moment then these editors are stacked using tabs. Editors allow the user to open, edit and save objects. Usually these objects are files, but editors can also be opened on other data sources such as for example a database object or some kind of input stream. Editors follow the classic open-modify-save paradigm much like file system based tools. When an editor is active it can add actions to the menu and the tool bar.
The editor area is designed to be the focal point of the workbench where the actual tasks are executed. The editor area is the same in every perspective. This means that if the user switches between different perspectives the currently open editors stay open and are visible in every perspective.

Views A perspective can have many views arranged around the outside of the editor area. A view provides information about an object the user is currently working with in the workbench. Views usually augment editors by reflecting some aspects of the currently active editor input or they augment other views by displaying information of an object selected in another view. Views have a simpler life cycle than editors: any action performed in a view should immediately be saved and affect the state of the workspace and underlying resources.

4 OMS Front-End Design

The process of finding a reasonable design for the new OMS front-end was influenced by two main goals.
One goal was to include the evaluation results presented in chapter 2. The OMS front-end that was developed during this diploma thesis provides a graphical user interface that can be uniformly applied to all OMS platforms supported by **OMS**[jp]. OMS Pro which is one of these platforms already offers its own user interface implemented in TCL/Tk. This interface was evaluated to identify strengths and weaknesses that led to a set of requirements listed in section 2.4. The new front-end was designed in order to maximise the number of satisfied requirements.
The second goal was to find a design that integrates as seamless as possible into the existing Eclipse framework. As covered in section 3.4, Eclipse has its own user interface paradigms. The aim was to respect these paradigms as far as possible in order to finally get a front-end that fits well into the Eclipse platform. This second goal was not always compatible with the first one. Therefore the Eclipse conventions were broken in some cases in order to meet the established requirements or in order not to lose beneficial functionality the new front-end should adopt from the OMS Pro GUI.
Section 4.1 describes issues that had to be solved in the design process and some proposals that were not realised in the final version because of the time available. The final design is presented in section 4.2.

4.1 Design Issues and Proposals

At first view the main window of the OMS Pro user interface and the Eclipse workbench look quite alike. The Eclipse workbench window though, can have several perspectives and the user is allowed to change the layout of these perspectives. The main Eclipse window can therefore adopt many different appearances. However, by comparing a predefined perspective

layout as presented in figure 3.4 to the main application window of OMS Pro (see figure 2.1) it is apparent that their structure is very similar in design. Both windows have a menu bar and a tool bar at the top. The right side of the window is used to display the project structure or the database structure respectively. At the bottom a console window can be found in both cases and the main area is used to open objects displayed in the right window component. Considering these facts it was obvious to reason that a new Eclipse-based front-end could easily be fitted into an own perspective of the Eclipse workbench.
However, at closer inspection not everything was as clear as it seemed and we had to take some major decisions concerning the object arrangement and the layout of the OMS front-end. These decisions also influenced the process of adding links between objects as described in section 4.1.2.

4.1.1 Layout and Object Arrangement

One component that we were sure about right from the start was a database explorer view that displays the contents of the databases present in the workspace similar to the Eclipse resource navigator. Besides this database explorer component it needed some consideration until we got to the final design. One of the main difficulties was to find a solution how to map the main application area of OMS Pro to the editor-view-paradigm in Eclipse. OMS Pro offers an area where the user can open multiple objects which can be viewed side by side or as overlapping windows. On the one hand this can be an advantage since the user can for example view a collection and a member of this collection at the same time. On the other hand the problem of this object area is that it tends to become overcrowded and quite complex with an increasing number of open objects. Therefore we intended not to adopt the concept of overlapping windows.
Eclipse offers an editor area which is designed to display a single editor at a time representing a file or in our case a database object. For every new editor window that is opened a new tab is added to the tab bar. If the user wants to have more than one editor window displayed on the screen he can tile the editor area arbitrarily by dragging editor windows with the mouse to a new location. Unfortunately this mechanism cannot be accessed programmatically using an API. The only way to use it would involve manipulations of internal Eclipse classes which is rather unreliable and which the Eclipse developers strongly advise against it. Thus we had to accept that the editor area normally displays only one object at a time.
We discussed several approaches to get around this problem by incorporating the views that can be arranged around the editor area. The following paragraphs describe the main choices that were discussed.

View to Display Objects An early idea was to use a view to display database objects such as collection members, type objects, methods and macros. Only collection objects should be opened in the editor area. A schematic perspective layout is provided in figure 4.1.
The advantage of this approach is that collections and objects are visible at the same time. The user can open several collections in the editor area. Each of the collections is represented as an editor and is accessible through a tab in the tab bar.
As described in section 3.4 views usually reflect aspects of the currently active editor or of

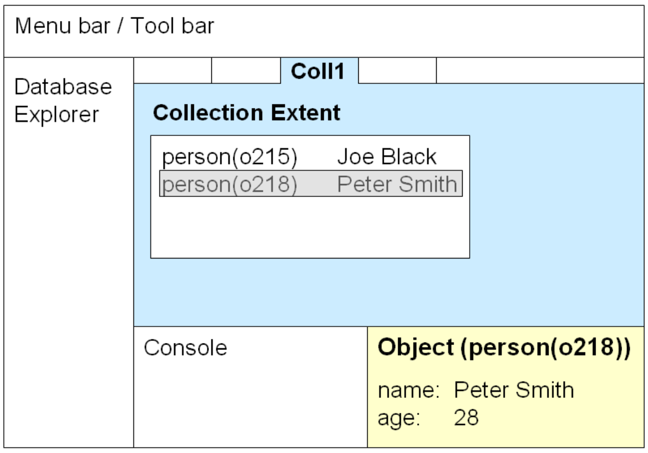

Figure 4.1: Schematic Perspective Layout using a View for Objects

objects selected in another view. As a consequence of this, a view can always display one object that is selected somewhere else in the workbench. If the user wants to open an object, he either has to select the object in the database explorer or he has to open the collection where the object is member. This means that switching between two objects belonging to two different collections also involves switching between these collections. This is rather circumstantial and we can assume that the user would like to have more than one object open at a time. Another disadvantage is the focus. An Eclipse convention says that the editor area should be the focal point of the workbench. Therefore views are typically placed marginally. Browsing and editing objects is a very frequent task while modeling and administrating a database. Having a view displaying these objects would force the user to focus on the margin instead of the centered editor area to a rather large extent of the working time.

View to Display Collections Another proposed solution was to invert the arrangement. Since collections are less often edited than objects, the idea was to display collections in a view while objects are opened in the main editor area. Figure 4.2 shows the possible layout. Using this solution the perspective would still profit from the fact that both collections and objects are visible at the same time. Moreover the user can open several objects in the editor area and easily switch between them. But there is again the disadvantage that a view can only display one object and therefore only one collection. This is neither what we had in mind since we did not want to restrict the browsing functionality by only allowing to view one collection at a time. A further disadvantage is that the presentation of a collection uses more space than a view normally does since some collections are binary or even ternary. The consequence is that the console window or for example an AQL window to query the database need to share the space with the collection view. This does not make much sense since the console should always be visible to display information and error messages.

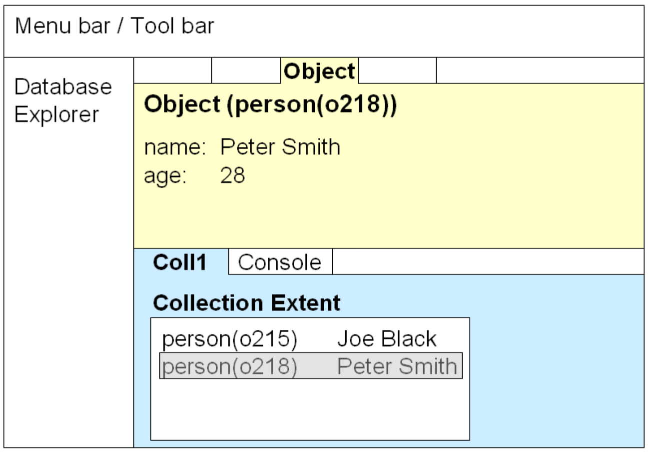

Figure 4.2: Schematic Perspective Layout using a View for Collections

Editor Area for all Objects A third discussed approach was to abandon the idea of using views and to display all objects in the editor area instead, regardless if it is a collection or another object. The space at the bottom of the window can be used for the console as shown in figure 4.3.

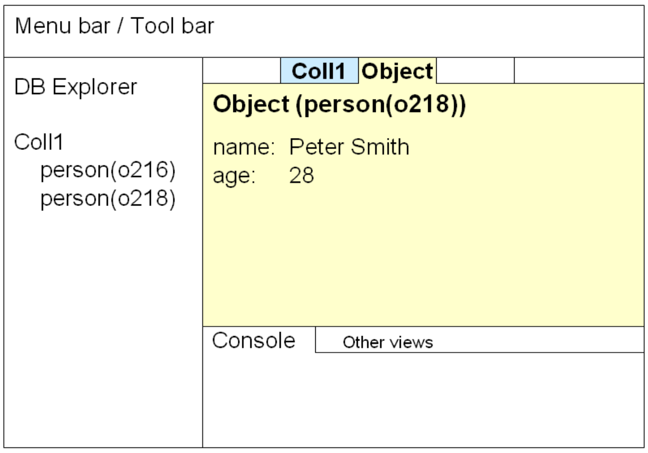

Figure 4.3: Schematic Perspective Layout using the Editor Area for Objects and Collections

The advantage of this solution is that the user can open as many objects and collections as he wants. The editor area becomes quite large if only a small console view is placed at the bottom of the window. If a collection is displayed it is useful to have enough space available since the user can access more collection members without scrolling. If an object is opened in the editor area, the space is not completely filled since only some attribute values are presented. But the object can be augmented by displaying context information in the remaining space.

The aim of context information is that the user can gain access to associated objects and collections directly by following links in the object editor. Displaying all objects in the editor area implicates that the user is not able to see more than one object at a time, but we assume that it is not so important since associated objects are directly accessible using the provided links.

After having discussed the advantages and disadvantages of these three approaches, we decided to take the third one since it seemed to be the most promising and also the most flexible approach concerning the layout of the perspective. If all objects are opened in the editor area the user is free to add whatever views around it and there are no constraints on the arrangement of the views.

4.1.2 Adding Links between Objects

The process of adding links between objects is one of the weaknesses of OMS Pro as described in section 2.3.2. Hence the aim was to improve the way of introducing links.

One idea was to display source and target collections, select an object from both collections and drag these objects into a binary collection. Since the editor area can only display one collection at a time we had to abandon this idea.

However, another PhD student of the group provided us with results from one of her former studies. She found that if a user intends to link two objects, it is more natural to establish a link starting from one object instead of taking two objects and connect them. Therefore we decided that the OMS front-end should adopt this technique. The idea is that the user can add a link starting either from the source or the target object. To establish the link the user only has to choose the second object and the object pair is automatically added to the binary collection. This technique avoids that the user has to open both objects and the binary collection as it is necessary using object fishing of OMS Pro.

4.1.3 Schema Editors

One of the established requirements was to offer graphical schema editors. They were planned to be part of the final user interface, but in the end they were not implemented due to time constraints and therefore not part of the final design. Nevertheless the following paragraph describes some design ideas how to integrate the schema editors into the OMS front-end since they are an important component to round out the user interface. All modeling functionality is provided in the OMS front-end version without the schema editors but they would support the modeling process by visualising the relationships between collections.

The three different schemas as described in section 2.2 could be integrated in a multi-page editor that opens in the editor area. Figure 4.4 shows how such an editor could look like.

The advantage of a multi-page editor is that the user does not have worry about choosing the right schema editor, but he can open the multi-page schema editor and easily switch between the pages until the appropriate schema editor is found. Further it is often the case that the user needs more than one schema editor to perform a task. For example the process of adding a new collection in the classification editor often involves the creation of a new object type in the type editor. Therefore it is convenient for the user to have all three schema editors bundled in one editor window.

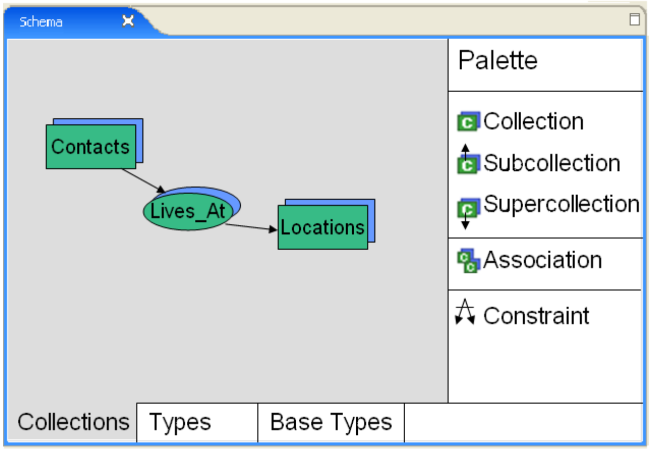

Figure 4.4: Proposal for a possible Schema Editor

To simplify the actual editing process, the schema editor could offer a design palette containing the graphical elements that can be added to the schema. The schema editor would be linked with other views as for example the database explorer.
The Eclipse Graphical Editing Framework (GEF) which is a powerful framework for visually creating and editing models could be a helpful tool to implement such a graphical schema editor. Basic information can be found in [5] and [14].

4.2 Final Design

Discussions of the design issues mentioned in section 4.1.1 led to a final design which is described and visualised in this section. The OMS front-end consists of an own perspective that is added to the Eclipse workbench. The perspective initially comprises three views, a database explorer presented in section 4.2.2, an AQL window described in section 4.2.5 and a console window (see section 4.2.6). A fourth view is opened every time a match operation is executed. A description of this view can be found in section 4.2.7. The editor area forms the focal point of the workbench and as stated earlier all objects that are opened are shown in this area using a corresponding editor (see section 4.2.3 and 4.2.4). Since the editor area is quite large, the empty space is used to present context information. The user can view associated information at first sight and navigate by following the provided links. Browsing the database becomes similar to surfing the internet since links to objects that might be relevant for the user are directly shown in the editor. Every time a link is opened a new editor window is added to the editor area including a new tab in the tab bar. This induced us to name this browsing technique ‘Tabbed Browsing’. The aim of this technique is that the user can work mainly task-driven. This means that the user can perform his work having in mind a high-level task that he wishes to accomplish. Since the system offers context information and links to associated objects, the user does not have to interrupt the task in order to find a certain object,

but he is able to use the guidance of the system and follow straightforward links to associated objects.

4.2.1 OMS Perspective

The core piece of the OMS front-end is an own perspective that is added to the Eclipse workbench. A screenshot of the OMS perspective is provided in figure 4.5.

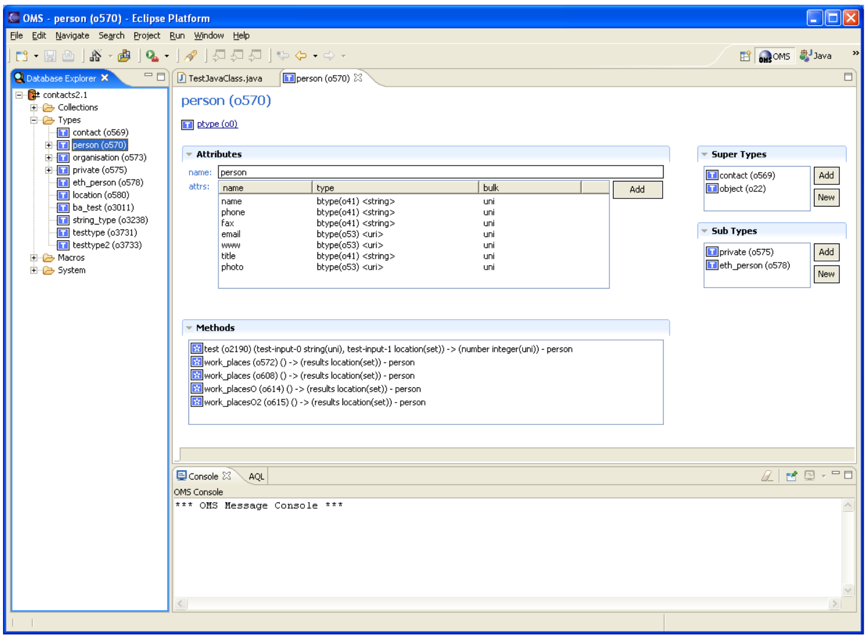

Figure 4.5: Eclipse Workbench, OMS Perspective

Springgay [11] advises to only add a perspective to Eclipse if it is really necessary. He fears that if every Eclipse plug-in uses its own perspective, the user would soon be overwhelmed by perspective choices. However, he considers that it is justified to add a perspective if there is a certain group of related tasks which would benefit from a predefined configuration of actions and views. For the OMS front-end this is the case. The process of modeling and administrating a database is completely unrelated to developing Java projects or Eclipse plug-ins and it therefore makes sense to integrate the numerous database tasks into one perspective. Furthermore working with OMS databases needs its own views such as the database explorer or the AQL window and by creating an own perspective these views can initially be added to the perspective layout in order to make all necessary tools available.
The user can open the OMS perspective using the 'Window' menu in the Eclipse menu bar or using the provided shortcuts in the 'Resource' and in the 'Java' perspective. When switching between perspectives the currently open editor windows stay open. Thus, if the user has several Java classes open and switches from the Java to the OMS perspective and vice versa, these editors are not closed. It may appear confusing to have Java files among database

objects, but it conforms to the Eclipse conventions and it allows to work with different sources without actually having to switch between perspectives. For a developer it might be useful to have a Java source file next to a database object if the Java application accesses an OMS database.

4.2.2 Database Explorer

The most important view of the OMS perspective is the 'Database Explorer'. It is the analogue of the 'Navigator' in the 'Resource' perspective or of the 'Package Explorer' in the 'Java' perspective. Considering OMS Pro, the database explorer combines the 'Collection' and the 'Macro' panel and enhances them by adding object types, collection members and methods. Figure 4.6 shows the database explorer.

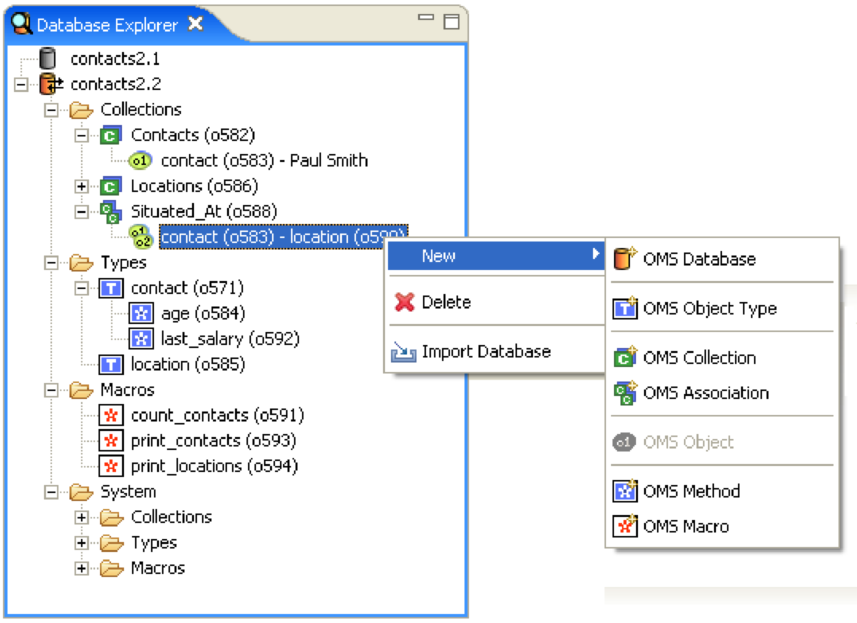

Figure 4.6: Database Explorer View

The database explorer shows all database instances that were either imported from the local file system or created directly within Eclipse. If a connection to one of the database instances is established, the content of this database can be browsed. Due to SICStus Prolog, the implementation language of OMS Pro, only one database connection can be open at a time. The explorer strictly separates user and system data which helps the user to keep track of the data. User as well as system data are categorised analogue into collections, types and macros. Types list their associated methods and collections provide access to their members. At the beginning of the design phase we were not quite sure if the database explorer tree should make the collection members available or not. Collections can have a large number of members and by expanding such collections in the tree, it can be rather difficult to overview

the database workspace and it uses quite a lot of scrolling activity. Nevertheless we decided to list the collection members since the editor area only shows one object at a time. Collection members and collections cannot be visible side by side in the editor area and therefore it makes sense to display the collection members in the database explorer tree to allow drag and drop operations and to complete the information in the database explorer.
Collection members are represented as strings comprising the type, the object id and an attribute value. By default the first attribute value of an object is taken, but the user has the possibility to choose a different attribute value for every collection. Displaying an attribute value for every collection member contributes to an improved usability by reducing the use of object ids. The user is able to choose a key attribute in order to find objects without having to know objects ids by heart or opening all objects until the desired one is found.
Every type of object introduces its own icon in the database tree which is used throughout the whole OMS perspective. Consistently used icons support the user to identify objects visually. The user can execute various actions in the database explorer that are gathered in a context menu. The context menu adapts according to the user selection in the tree. An important menu entry is the 'New' menu which enables the user to create new databases and new database objects. In most of the cases choosing a 'New' action launches a wizard. As an example, the wizard to create new OMS databases is shown in figure 4.7.

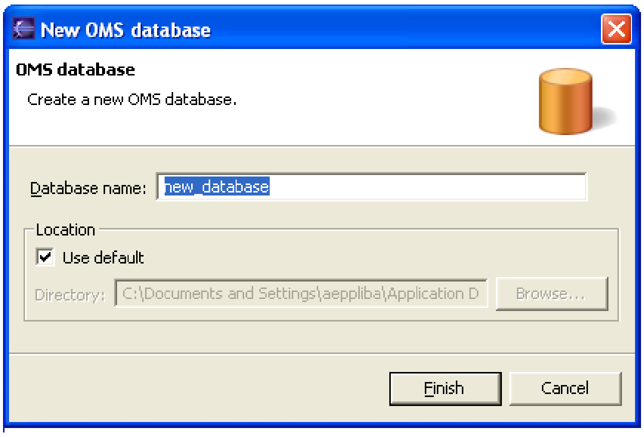

Figure 4.7: Wizard to Create a New OMS Database

The wizards are necessary since the user has to indicate preliminary information in order to enable a precise object creation. The entry fields in these wizards are pre-filled depending on the selection in the database explorer. This helps the user to efficiently create new objects without having to fill in large forms. For example to create a new association between two collections, it is possible to select two collections in the database explorer and launch the wizard. The user only has to indicate which of the collections should be the source collection and the association can be created. If the user wants to create database objects which do not need any preliminary information as for example a new macro, wizards are omitted to simplify the process of creating new objects.

4.2.3 Collection Editor

If the user double clicks on a collection in the database explorer, an editor window as shown in figure 4.8 is opened in the editor area. The collection editor is a multi-page editor which means that it can contain various pages accessible by means of a tab bar at the bottom of the window.

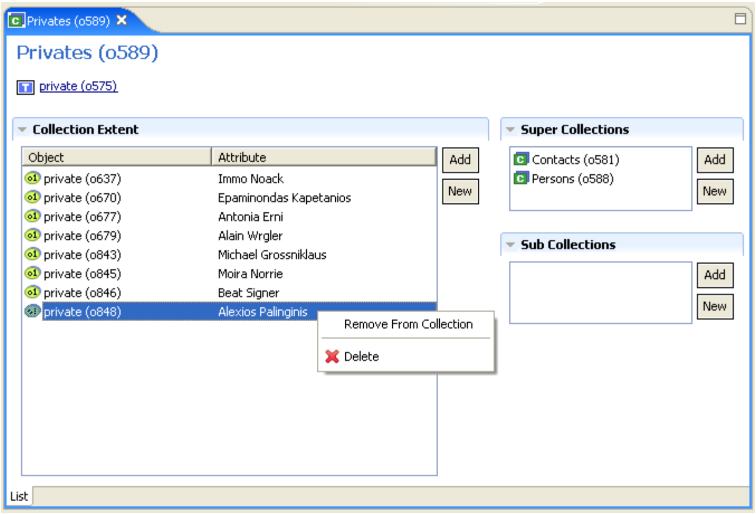

Figure 4.8: Collection Editor

The final version of the collection editor only contains a single page, but we still chose to use a multi-page editor to make the editor easily extensible. Additional pages could contain more context information or associated DDL (Data Definition Language of OMS) statements. In the final version the single page displays the collection extent as a table where the user can add and remove members. The right hand side of the editor page shows all super- and subcollections. Below the title a link to the member type of the collection is provided.
The style of the collection editor is based on a new Eclipse 3.0 feature called ‘Eclipse Forms’. Eclipse Forms is a plug-in that offers a set of custom widgets and other supporting classes that were before used as internal Eclipse classes. These classes form a toolkit to create ‘Web-like’ user interfaces by extending SWT in order to get the desired behavior. SWT controls can typically appear either in traditional dialogs (message boxes, dialogs, wizards, preference pages) or in content areas used for views and editors. If the controls are added to dialogs they use colors and fonts as provided by the operation system and the goal is to fill a rather small dialog area. Controls in views and editors are supposed to fill the entire content area, they should scroll their content and the colors and fonts should be those provided by the system for use in the content area. Eclipse Forms are designed to meet these requirements by offering a toolkit that manages among other things colors and a factory which creates basic SWT controls in order to fit into the form context. The result looks very much like forms in HTML browsers. Therefore the form toolkit comprises some additional custom controls used to add hyperlinks, image hyperlinks and expandable sections to the form. More information about

Eclipse Forms can found in [13], a draft of a programming guide that will eventually move to the official Eclipse help.

4.2.4 Object Editor

Concerning the means of design the object editor is very similar to the collection editor. It is also a multi-page editor using Eclipse Forms. The object editor is used to represent all kind of database objects apart from collections. The screenshot provided in figure 4.9 shows the representation of a collection member. The title of the page as well as the tab label display an attribute value of the represented object in addition to the object id. This helps the user to easily get back to an already open editor without having to remember the object id.

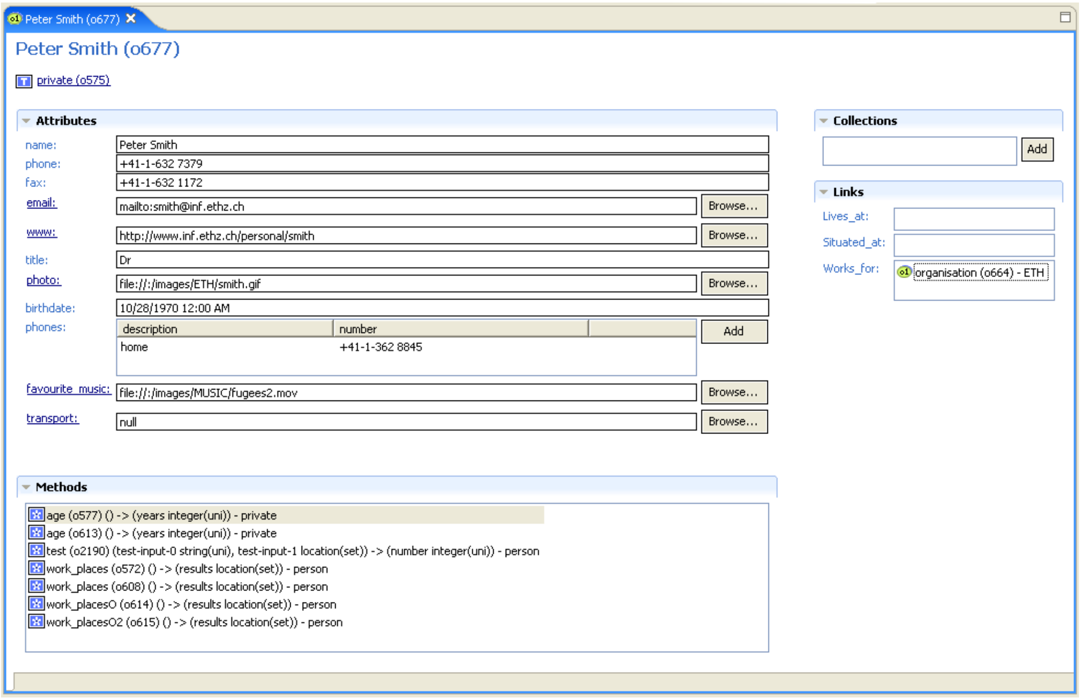

Figure 4.9: Object Editor

Depending on the kind of object the editor offers different information sections. The 'Attributes' section is common to all kinds of objects whereas the 'Links' section is special for this type of object. For collection members the 'Links' section offers valuable context information that helps the user to immediately understand the context of an object.
By convention editors in Eclipse usually follow a classical open-modify-save paradigm. In chapter 2 we stated that the commit/rollback paradigm is one of the strengths of OMS Pro and should thus be adopted in the OMS front-end. We therefore decided to break the conventions. The collection editor as well as the object editor do not need any save operations. Any changes made to database objects are immediately reflected in the database.

4.2.5 AQL Window

In addition to the database explorer view the OMS front-end offers an AQL view which can be used to query databases using the own query language of OMS, called AQL. In the initial perspective layout the AQL view is placed in the console area behind the console window. Figure 4.10 shows how the AQL window looks like.

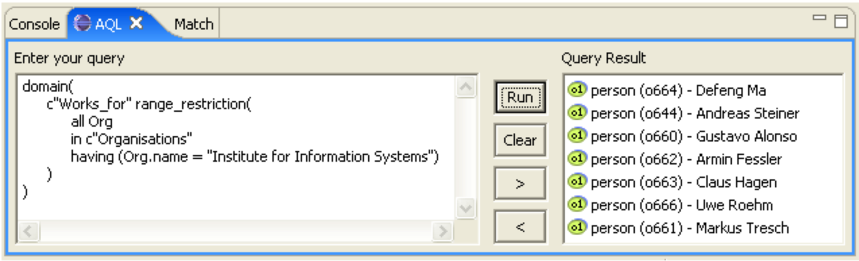

Figure 4.10: AQL View

On the left hand side of the window the user can enter the query. After the execution of the query the result is displayed in the table on the right side. If the result comprises objects, double clicking them will open the object in the editor area. A history functionality enables the user to revert to previous executed queries.

4.2.6 Console Window

The Eclipse workbench provides a console view which can display several consoles. The OMS front-end uses its own OMS console which is added to the Eclipse console view when the OMS perspective is activated. The OMS console displays error messages in red and information messages in blue as it is presented in figure 4.11.

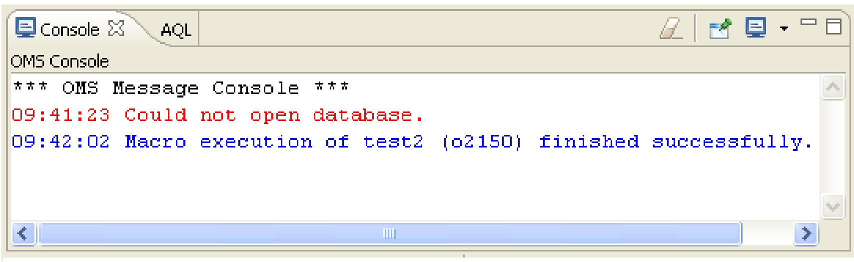

Figure 4.11: OMS Console

4.2.7 Match Window

The OMS front-end offers a fourth view, the ‘Match’ view. The view can be compared to the ‘Search’ view in the ‘Java’ perspective of Eclipse. The view is not initially shown in the OMS perspective, but the perspective contains a placeholder which defines where to open the view if it is needed. The match view is opened every time the user wants to retrieve database objects that match a certain object type using the ‘Match’ operation in the context menu of the database explorer.
The retrieved objects are listed in the result table of the match view. Figure 4.12 shows two match view windows. The window on the left hand side displays all structured types that are part of the database whereas the right match window shows all existing triggers. All objects can be opened and edited with a double click.

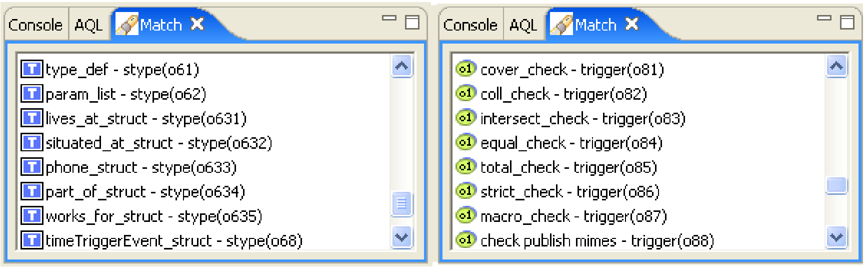

Figure 4.12: Two Match Windows

5 OMS Front-End Implementation

The OMS front-end is implemented as an Eclipse plug-in using the Eclipse plug-in mechanism as described in section 3.3. The front-end plug-in contributes to the Eclipse workbench which is the user interface of Eclipse by extending various extension points offered by the workbench.
If the OMS front-end is installed, Eclipse adds a plug-in directory to the Eclipse installation directory. The directory name is **ch.ethz.globis.omsjp.omsbrowser** which is the global identifier of the OMS plug-in. The structure of the plug-in directory is similar to those of all other plug-ins and contains the following files and folders:

- Plug-In XML File
  The plugin.xml file is a file in XML format that describes the OMS plug-in and how it integrates into Eclipse.

- Icons Folder
  The plug-in directory contains typically an 'icons' or 'images' subdirectory to place image files that ship as part of the plug-in. In the OMS plug-in the subdirectory is called 'icons' and it contains various image files that are used for the graphical design of the plug-in. The 'icons' folder is referenced by the plugin.xml file and by various plug-in classes.

- Library Folder
  The OMS plug-in accesses OMS databases by means of the **OMS**^{jp} library. The **OMS**^{jp} library consists of a JAR file called omsjp.jar. To work with OMS Pro, **OMS**^{jp} needs an appropriate driver which is stored in a JAR file called omspro.jar and a Prolog library called jasper.jar. The plug-in directory contains therefore a library folder called 'lib' where these three JAR files are placed.

- OMS Jar File
  The actual code of the plug-in is stored in a JAR file as well. Typically the JAR file is named for the last segment in the plug-in's identifier, but it could basically have any name, as long as the name is declared in the plug-in manifest file. For the OMS plug-in the JAR file is called OMSBrowser.jar which in fact corresponds to the last segment of the plug-in identifier.

The plug-in code is divided into twelve Java packages containing a varying number of Java classes. All package names are prefixed by the string **ch.ethz.globis.omsjp.omsbrowser**. In the UML diagrams appearing in this chapter classes and interfaces in grey are provided by the Eclipse platform whereas those in red are own classes and interfaces. Sections 5.2 until 5.13 explain all packages in more details whereas section 5.1 provides information about the plug-in manifest file.

5.1 Plug-In XML Manifest

As stated in chapter 3 every plug-in has exactly one plug-in manifest file. The manifest defines various high-level aspects so that the plug-in does not have to load until some functionality is required. It contains four major parts which are described in the following paragraphs providing some XML snippets. The whole plug-in manifest file can be found in the appendix A.

Plug-In Declaration Section The declaration contains the plug-in identifier which is designed to uniquely identify the plug-in. Usually the same naming convention as for Java packages is applied. Therefore the identifier of the OMS front-end corresponds to the package prefix. Further the declaration specifies the name, the plug-in provider and the version. These values are human-readable text and are not required to be unique. The last attribute in the declaration indicates the plug-in class which is described in section 5.2. Listing 5.1 shows the plug-in declaration of the OMS plug-in.

```
<plugin
   id = "ch.ethz.globis.omsjp.omsbrowser"
   name = "OMSBrowser Plug-in"
   version = "1.0.0"
   provider-name = "ETH Zurich, Global Information Systems Group"
   class = "ch.ethz.globis.omsjp.omsbrowser.OMSBrowserPlugin">
```

Listing 5.1: OMS Plug-In Declaration

Runtime Section The 'runtime' section of the plug-in enumerates the libraries that contain the plug-in code. For each library the export attribute specifies which classes are accessible to other plug-ins. This is necessary since Eclipse imposes more restrictions on plug-in

interaction than in a typical Java application. Each plug-in has its own class loader, restricting the visibility to code specified in the plug-in manifest. This means that a class X in one plug-in which is in the same package a another class Y in a required plug-in cannot access the ‘public’ method in class Y. The compilation process finishes without errors, but if the code is executed in the Eclipse framework, the class loader will restrict the access and throw an exception. Since the OMS front-end should be extensible to add future components, all libraries are exported to allow another plug-in to access them.

Dependencies Section The ‘requires’ section of the plug-in manifest indicates all plug-ins that the OMS plug-in is dependent on. This information is needed since the plug-in class loader must know which plug-ins will be visible to the OMS plug-in during execution. It is also possible to require a designated version of a plug-in. If the OMS front-end is activated, all required plug-ins are activated transitively if they have not been loaded before.

Extensions Section The OMS front-end declares various extensions offered by the Eclipse workbench to contribute own components to the user interface. Listing 5.2 shows the view extension that is used to define the database explorer view. The extension specifies among other things the name and the icon of the view, the category the view belongs to and the main class which implements the view.

```
<extension
   point="org.eclipse.ui.views">
   <category
      name="OMS"
      id="ch.ethz.globis.omsjp.omsbrowser">
   </category>
   <view
      name="Database Explorer"
      icon="icons/database_explorer.gif"
      category="ch.ethz.globis.omsjp.omsbrowser"
      class="ch.ethz.globis.omsjp.omsbrowser.views.DBExplorerView"
      id="ch.ethz.globis.omsjp.omsbrowser.views.DBExplorerView">
   </view>
</extension>
```

Listing 5.2: OMS Plug-In Database Explorer View Extension

Similar extension declarations exist for all views and editors, the perspective as well as for wizards and the preference page.

5.2 Package OMSBROWSER

The **omsbrowser** package contains a single class called **OMSBrowserPlugin** which is the plug-in class specified in the plugin.xml declaration section. The class offers methods for the startup and the shutdown of the plug-in which are called by the Eclipse runtime environment.

Once plug-ins are loaded into the memory, they are never unloaded until the Eclipse workbench is closed. Therefore the shutdown method is only called when the user choose to leave Eclipse. To preserve the state of the OMS perspective across multiple platform sessions, the plug-in class contains methods to save and restore objects displayed in the database explorer based on a memento mechanism. Mementos are designed to save and restore objects in a form that they can be stored persistently in the file system. They are able to skip objects in the restoring process if an appropriate class is not available anymore or to restore objects from the provided data even though the class for the object is different from the one that was originally saved. Mementos map arbitrary string keys to primitive values and they allow to have other mementos as children. The **OMSBrowserPlugin** class uses a memento based on XML as external storage format. If the user closes the Eclipse workbench, the state of the database explorer is saved to a file called ‘oms.xml’. The file is placed in the Eclipse runtime workspace located in the local file system. Listing 5.3 shows a sample ‘oms.xml’ file which represents two database instances in the workspace. The ‘Contacts’ database was connected when Eclipse was closed and therefore the connection will be reopened when Eclipse is started again.

```
<?xml version="1.0" encoding="UTF-8" ?>
<oms>
   <database
      connected = "false"
      name = "Phonebook"
      path = "C:/Program Files/omspro2.1/demo/phonebook.oms" />
   <database
      connected = "true"
      name = "Contacts"
      path = "C:/Program Files/omspro2.1/demo/contacts.oms">
         <collection attr="friends" id="o1261" />
         <collection attr="name" id="o1389" />
         <collection attr="name" id="o592" />
         <collection attr="street" id="o596" />
         <collection attr="phone" id="o600" />
         <collection attr="characteristics" id="o242" />
   </database>
</oms>
```

Listing 5.3: Sample OMS.XML File

Another functionality of the **OMSBrowserPlugin** class is the initialisation of the OMS preference page to configure the **OMS**jp driver. The implementation of the preference page is described in section 5.12.

Furthermore, the **OMSBrowserPlugin** class maintains an instance of the class **java.util.HashMap** as image cache and it offers a public method to retrieve images used in the user interface of the OMS front-end.

5.3 Package MODEL

The database explorer (see section 4.2.2) is designed to display a hierarchical list of OMS database objects in a parent-child relationship. From a conceptual point of view it is quite similar to the 'Navigator' in the 'Resource' perspective or the 'Package Explorer' in the 'Java' Perspective. Both the navigator and the package explorer use the Eclipse **IResource** class and its subtypes as internal model. The **IResource** class is the generic class for files and folders which form per se a hierarchical structure. In order to map the OMS database objects into the database explorer tree structure, it was necessary to build a custom internal model based on a parent-child relationship. The final model is shown in figure 5.1 presenting classes and the most important public methods.

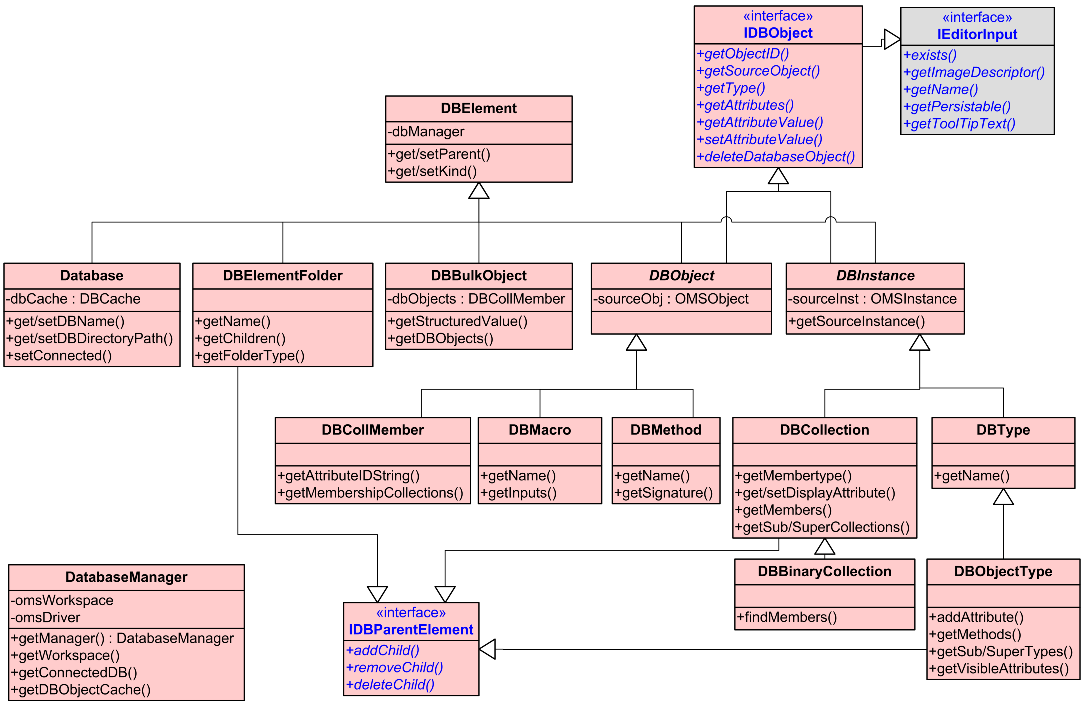

Figure 5.1: Model Classes

The **DBElement** class comprises all objects that can be displayed in the database explorer tree offering methods to set and get the parent element. Using the **setKind** method, tree objects are classified into system and user data objects. The **DBElement** class is abstract and does not have a physical manifestation in the tree. However, it has five subclasses representing different tree objects. Instances of the **Database** class build the top-level elements in the tree standing for OMS databases stored in the local file system. The **Database** class has a reference to the **DBCache** class. The cache enables the database to save and retrieve created tree objects in a rather efficient way since the **DBCache** uses instances of the class **java.util.Hashtable** class to manage the tree objects.

On the next hierarchy level the tree displays folder objects to reasonably structure the database contents. These folders are represented by the class **DBElementFolder** which internally stores its child objects.
The **DBInstance** and the **DBObject** class comprise all database object that corresponds one-to-one to an OMS object in the OMS database system. Therefore these two abstract classes implement the **IDBObject** interface offering methods common to all OMS objects. The classes wrap **OMS**^{jp} objects, namely instances of **OMSInstance** and **OMSObject** respectively.
The **DBObject** class provides three concrete subclasses for collection members, macros and methods: **DBCollMember**, **DBMacro** and **DBMethod**. These classes offer in addition to the common database object methods, specific methods applicable only for this kind of object.
The **DBInstance** class comprises types and collections objects. In **OMS**^{jp} collections, types and associations are subtypes of the **OMSInstance** class which represents an instance of the underlying **OMSObject**. The **DBInstance** class holds therefore a reference to the **OMSInstance** class. Collections are classified into unary and binary collections which are mapped to the class **DBCollection** and its subclass **DBBinaryCollection**. The **DBType** class comprises all possible types available in OMS whereas the **DBObjectType** class is a specialised class for object types offering additional methods. Since the user should be able to open all these OMS objects in the Eclipse editor area, the **IDBObject** interface inherits from the **IEditorInput** interface enabling these objects to be opened by an editor.
The last subclass of **DBElement** is the **DBObjectBulk** class which stands for binary or multi-valued collection members. These objects do not correspond to any OMS object and therefore this class does not implement the **IDBObject** interface, but subclasses the **DBElement** class directly. Internally it holds references to the underlying instances of **IDBObject**.
Having this class hierarchy allows us to map the OMS database objects into a hierarchical tree structure so that every object knows its parent object and all child objects. Apart from the **Database** class, three classes in the model can act as parent objects. All other classes are tree leaves. The classes that are possible parent objects implement the **IDBParentElement** interface offering methods to add, remove and delete child objects. The **Database** class does not need to provide these methods since its only children are instances of the class **DBElementFolder** which cannot be deleted or removed.
Providing a hierarchical class structure for the tree viewer in the database explorer is one thing, but all these objects and the whole OMS front-end need access to the underlying OMS databases using **OMS**^{jp}. The connection handling is done by the **DatabaseManager** class which is one of the central classes in the OMS plug-in. The **DatabaseManager** class uses a Singleton pattern which means that only one instance of the class is available throughout the running time of the OMS plug-in. The database manager holds instances of all databases in the workspace and is therefore the parent object of the **Database** class. Further it initialises the OMS driver and the OMS workspace when the user starts working with database objects and it is responsible for presenting the OMS console when the OMS perspective is activated. The database manager class offers methods to import, create, open, close, delete, commit and rollback OMS databases and it verifies that at most one database connection is open at a time. Moreover, the class is responsible for managing listener classes implementing the **IDBElementListener** interface and notifies them if a **DBElement** object changes.

5.4 Package MODEL.EVENT

The **model.event** package offers a listener interface, called **IDBElementListener**. It contains methods which are called if an object of type **DBElement** is added, removed or changed or if the user chooses a different display attribute for a collection. Classes implementing the interface should register itself using the **addDBElementListener** method offered in the **DatabaseManager** class to make sure that it receives all notifications.
Event notifications concerning changes of **DBElement** objects use an own event object class called **DBElementEvent**. **DBElementEvent** inherits from the **java.util.EventObject** class and takes a **DBElement** as source event object in the constructor.

5.5 Package PERSPECTIVE

The Eclipse user interface defines the extension point **org.eclipse.ui.perspectives** to add a new perspective to the Eclipse workbench. The OMS perspective contributes to this extension point by providing an XML declaration in the plug-in manifest. The declaration contains the identifier of the perspective, its name and icon and the perspective class which has to implement the **IPerspectiveFactory** interface. The factory class is called **OMSPerspectiveFactory** and implements the single method defined in the **IPerspectiveFactory** interface, **createInitialLayout**. This method specifies the initial page layout for the perspective, namely where to put which view, how to place the editor area and what shortcuts to add in the various menus and tool bars. The factory class is only used to define the initial layout of the perspective when the perspective is opened. After that the class is discarded and the perspective is restored exactly the way as it was when Eclipse was closed.
OMSPerspectiveListener is the second class in this package, implementing the **IPerspectiveListener** interface. Its method **perspectiveActivated** is called whenever the user opens a new perspective or switches to an already open perspective. The **OMSPerspectiveListener** class checks if the OMS perspective was activated and if so, it reactivates the OMS console in the console view.

5.6 Package VIEWS

Views are part of the Eclipse workbench. From a developer's point of view it is necessary to specify the workbench more precisely and clarify some Eclipse terminology.
The term used to represent the entire Eclipse user interface is workbench. The workbench itself has no physical manifestation but it is displayed in one or more workbench windows. These basic top-level windows make up an Eclipse application having a menu and tool bar on the top and the status line at the bottom of the window. The main body of a workbench window between the tool bar and the status line is represented by the workbench page, which in turn is made up of workbench parts. Workbench parts come in two varieties, namely views and editors. The initial size and orientation of the parts in the page are determined by the perspective. Parts interact with the rest of the window via their site. The site is not a visible entity but simply an API mechanism to separate the methods that operate on the view from the methods that operate on controls and services outside the view.

5.6.1 Database Explorer View

The database explorer view is one of the three views that belongs to the OMS perspective. Similarly as the perspective extension point, a view extension point (**org.eclipse.ui.views**) is offered by the Eclipse workbench. The database explorer extends this point to register itself as an Eclipse view. Views must implement the **org.eclipse.ui.IViewPart** interface. Typically, views subclass the **org.eclipse.ui.part.ViewPart** class and are thus indirectly subclasses of **org.eclipse.ui.part.WorkbenchPart**, inheriting much of the behavior needed to implement the **IViewPart** interface. The **DBExplorerView** class is the main class of the database explorer and subclasses **ViewPart**. The **createPartControl** method overrides the method specified in the superclass. It is a callback method creating all controls comprised in the view. Figure 5.2 shows how the database explorer view integrates into the existing Eclipse framework. Classes and interfaces in grey are provided by the Eclipse platform.

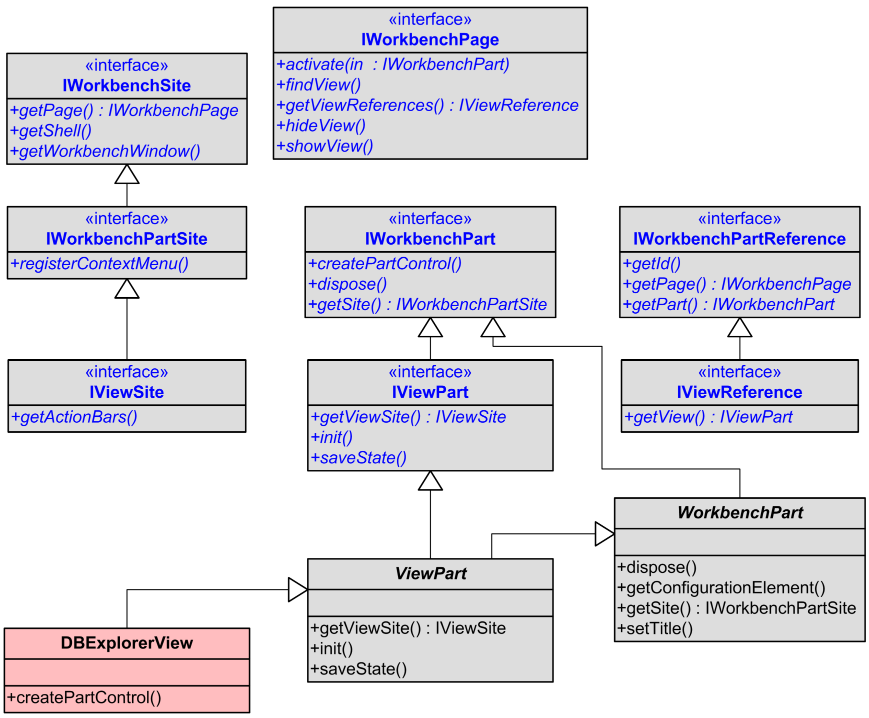

Figure 5.2: View Part Integration into the Eclipse Workbench

Views are part of a view site (**IViewSite**), which is part of a workbench page (**IWorkbenchPage**). In order to enable lazy instantiation, the **IWorkbenchPart** holds on to instances of **IViewReferences** rather than to the **IViewPart** itself. This permits that views can be referenced without actually loading the plug-in defining the view.
The database explorer uses the JFace UI toolkit (see section 3.2). The object tree in the database explorer is a JFace **TreeViewer** which wraps an instance of the SWT **Tree** widget. A tree viewer displays a hierarchical list of objects in a parent-child relationship. **TreeViewer**

is a subclass of **StructuredViewer** which represents an abstract base implementation for structure-oriented viewers like tree, list and table viewers. In general viewers are model-based adapters on a SWT widget which accesses its model by means of a content provider and a label provider. The classes used to implement the database explorer view are presented in figure 5.3.

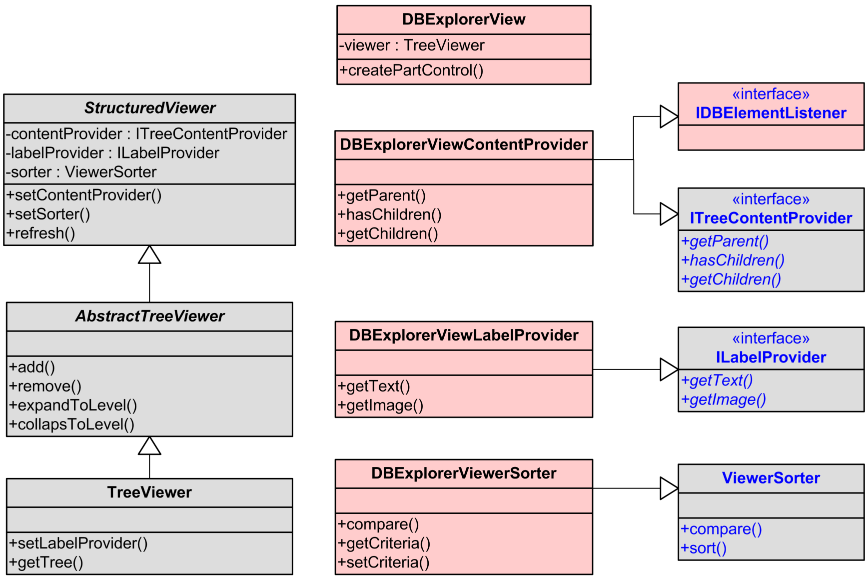

Figure 5.3: Database Explorer View Classes

The database explorer displays database objects and thus uses the model described in section 5.3. To access the model, the tree viewer uses the **DBExplorerViewContentProvider** which is responsible for extracting the appropriate objects from an input object, in our case the **DatabaseManager** which is the parent object of all databases in the workspace. Having the raw objects, the **DBExplorerViewLabelProvider** is called to determine the image and the label that is actually displayed in the tree. Optionally the tree viewer in the database explorer makes use of the **DBExplorerViewSorter** which sorts the tree objects according to defined sort criteria before they are displayed.
If the content of the database manager or any database object change, methods specified in the **IDBElementListener** interface are called. The **DBExplorerViewContentProvider** implements this interface and is thus responsible for reacting to any changes and updating the tree viewer.

5.6.2 AQL View

The AQL view integrates into the Eclipse platform in the same way as the database explorer. Its implementation comprises one class called **AQLView** which subclasses the **ViewPart** class in the Eclipse framework. Calling the **createPartControl** method creates the main panel for the AQL view consisting of a SWT **Text** entry, some instances of the SWT **Button** class

and a JFace **TableViewer** which is a subclass of the **StructuredViewer**. The table viewer is used to display query results by means of a content and a label provider. The **AQLView** class has an inner class called **ResultContentProvider** which implements the **IStructuredContentProvider** interface. This interface consists of a single method **getElements** to extract the table elements from an input object. In the **AQLView** the input object is an instance of the class **OMSValue** representing the query result. As label provider a base implementation of the **ILabelProvider** interface provided by the Eclipse framework is used. The **AQLView** class holds a private instance of the **ArrayList** class to store query strings that were executed in the past.

5.6.3 Match View

The match view is a third view that is implemented analogue to the other views. Its main class **MatchView** subclasses **ViewPart** and contains an inner class as content provider for the table that displays the found database objects. If the match action is executed and the view is currently closed, then the **MatchAction** class located in the **action** package makes sure that an instance of the match view is opened. The OMS perspective contains a placeholder for the match view behind the AQL view which defines the location of the view when it is opened.

5.6.4 Console View

The console view differs from the other views since the view itself is implemented and offered by the Eclipse framework. The OMS console is thus an extension of the existing console view. The implementation of the OMS console view comprises a single class **OMSConsoleManager** which is responsible for adding an OMS console page to the existing console using the **addConsole** method of the Eclipse **ConsolePlugin**. The OMS console is based on an instance of type **MessageConsole** and offers methods to write information and error messages to the screen.

5.7 Package EDITORS

The Eclipse platform offers some basic editors such as a standard text editor as well as highly specialised multi-page editors such as the plug-in manifest editor. The OMS front-end cannot profit from the available editors since a text editor is not sufficient to represent database objects and the plug-in manifest editor is too specialised. Therefore the OMS plug-in provides its own editors to view database objects.

Editors integrate into the Eclipse workbench in a similar way as views. Figure 5.4 is thus analogical structured as the view part classes presented in figure 5.2. The difference is that an editor class has to implement the **IEditorPart** interface instead of the **IViewPart** interface as views do and editors are contained in an **IEditorSite**. The **IWorkbenchPage** class holds on to instances of the **IEditorReference** class rather that the editor itself to allow lazy initialisation. The methods presented in the workbench classes are those used in connection with editors.

As apparent from figure 5.4 the OMS front-end uses two editors which are multi-page form editors. The object editor is described in more details in section 5.8 whereas more to the

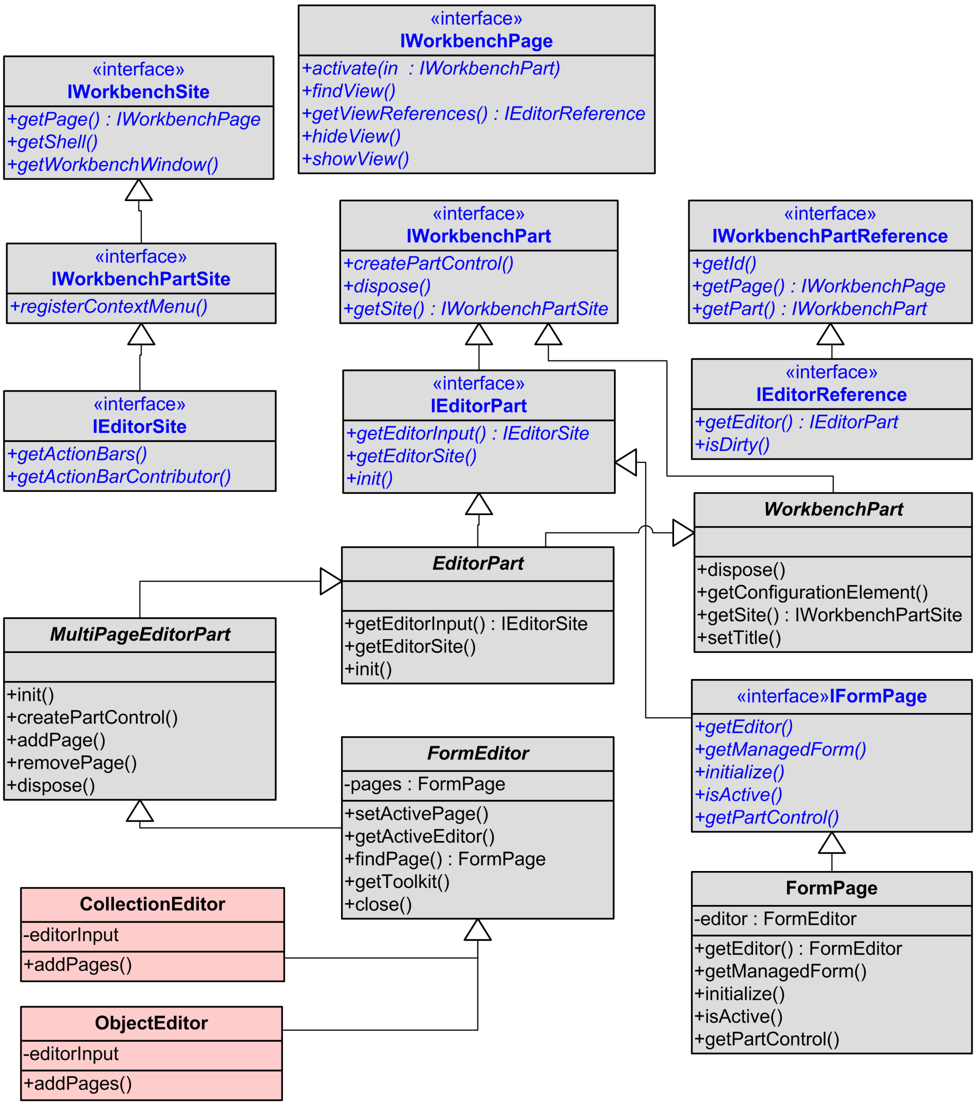

Figure 5.4: Editor Part Integration into the Eclipse Workbench

collection editor can be found in section 5.9. To construct these editors the OMS plug-in provides some kind of framework offering different components that can be reused in the collection and object editors as well as in dialog windows to add new values. Figure 5.5 provides an overview of the graphical components that are used to construct an editor page.

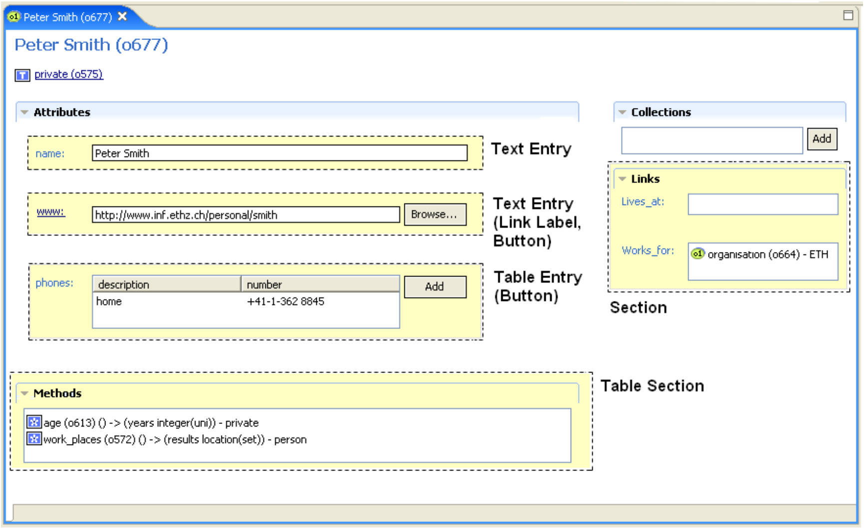

Figure 5.5: Building Blocks of an OMS Editor Page

An editor page can have various sections whereas each section groups information that belongs together. The Eclipse platform offers a base implementation of a section, called **SectionPart**. The **SectionPart** class as well as the **IFormPart** interface it implements are located in the **org.eclipse.ui.forms** package. A section has a title and can be in an expanded or collapse state. The OMS plug-in offers a **DBEditorSection** and an even more specialised **DBEditorTableSection**. The **DBEditorTableSection** is preset having a JFace table viewer as main component of the section with the possibility to add buttons to the right side of the table. The classes that belong to the framework are presented in figure 5.6
If a section is more complex than just displaying a table viewer with buttons to the left, the **DBEditorSection** can be used by adding the desired 'form entries' manually. Form entries are reusable components of the OMS plug-in editor framework that can be added to form editors. An entry can either be a text entry (class **DBTextEntry**) or a table entry (class **DBTableEntry**), both subclassing the abstract **DBFormEntry** class. An instance of the **DBTextEntry** class consists horizontally from left to right of an optional label, a SWT text field and an optional button. The **DBTableEntry** is similar, offering a JFace table viewer instead of a text field and more than one button to the right.
In order to add actions to the buttons or a hyperlink instead of a label, the framework provides the **IEditorFormEntryListener** interface. Classes implementing this listener interface can be associated with the entries to react to certain kinds of events. The **EditorFormEntryAdapter** class serves as a base implementation of the **IEditorFormEntryListener** interface.

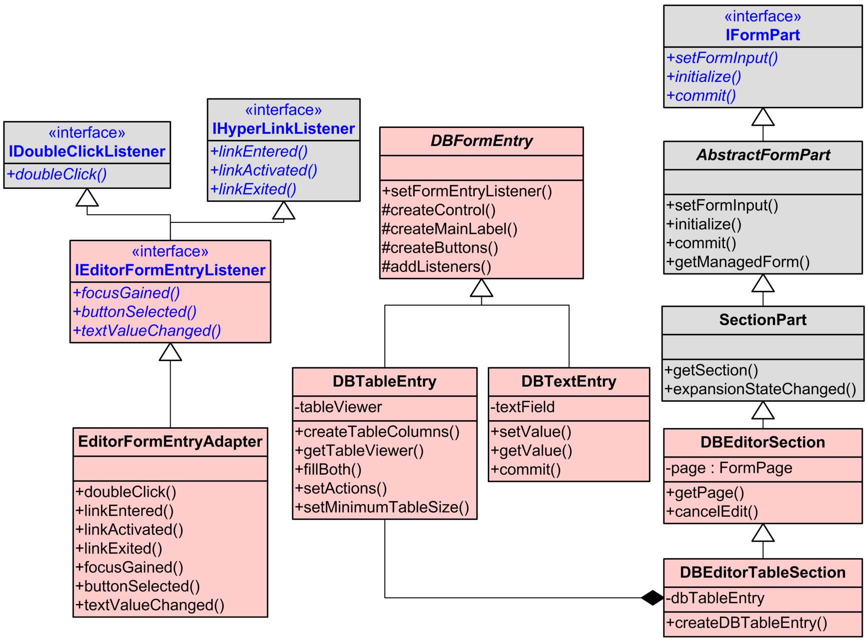

Figure 5.6: Reusable Form Editor Components

5.8 Package EDITORS.OBJECT

The OMS plug-in offers two different editors. The collection editor is used to display collection objects, all other database objects are shown in the object editor. The object editor (class **ObjectEditor**) is a form editor and therefore extends the Eclipse **FormEditor** class. It is a multi-page editor containing a single page, namely the **ObjectAttributePage**. As described in section 5.7 the OMS plug-in offers generic section components that can be added to the editor pages. The object editor is constructed of different sections, depending on the kind of database object it represents. Figure 5.7 provides an overview of all object editor classes.
The **ObjectAttributeSection** displaying the attributes of a database object is common to all database objects shown in the object editor. Type objects offer additionally the **ObjectMethodSection** and the **SubTypeSection** and **SuperTypeSection** to present sub- or supertypes respectively. Collection members offer the **ObjectCollectionSection** presenting all membership collections, the **ObjectMethodsSection** and the **ObjectLinksSection** to show related objects. All sections implement the **IDBElementListener** interface to be able to react to any events.
The **ObjectMethodSection**, the **SuperTypeSection**, the **SubTypeSection** as well as the **ObjectCollectionSection** display simple tables and extend therefore the **DBEditorTableSection** class. The **ObjectLinksSection** uses several **DBTableEntry** instances to display several link tables whereas the **ObjectAttributeSection** is the most complex one. It displays the different attribute values and depending on the attribute type. For each attribute type the form entry has to adopt a different shape. Therefore the **DBValueEntry** class exists which is a helper class

knowing how each attribute depending on its type has to be displayed. Some dialog classes as for example the **AddValueDialog** class reuse the **DBValueEntry** class to offer appropriate forms to allow the user to add attribute values.

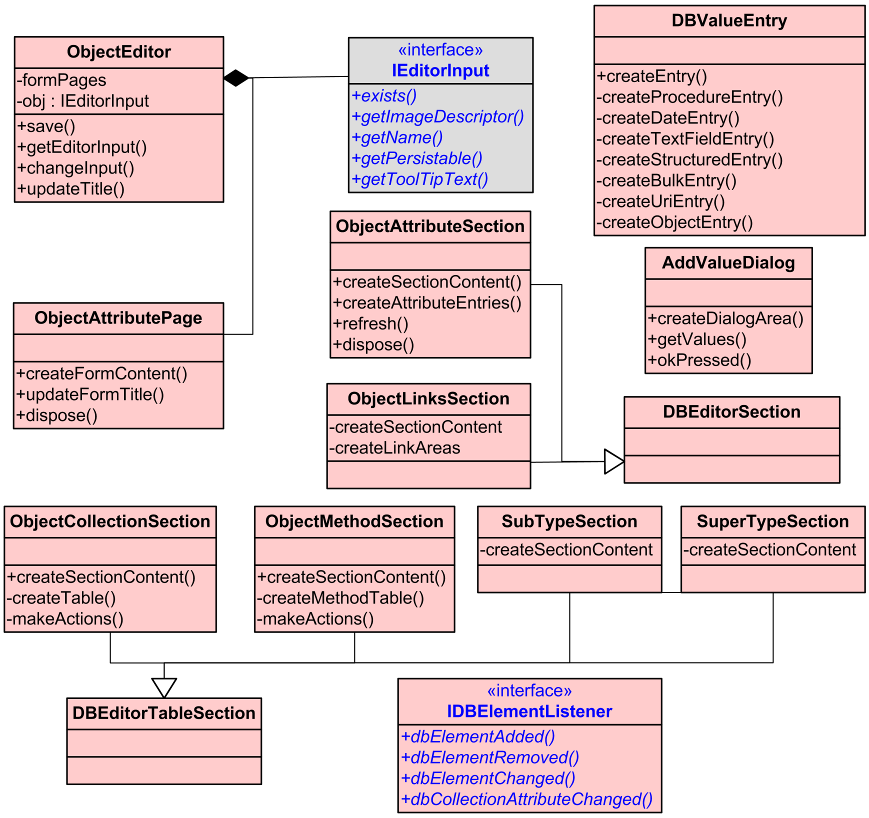

Figure 5.7: Object Editor Classes

5.9 Package EDITORS.COLLECTION

The collection editor is used to display collection objects. Its structure is analogue to the object editor. The **CollectionEditor** class is the main editor class extending the **FormEditor** class. It has a single page associated with it, namely the **CollectionListPage** which consists of three sections. The main section is the **CollectionListSection** presenting the collection extent in a JFace table viewer. The **SubCollectionListSection** and the **SuperCollectionListSection** offer sub- and supercollections respectively. The sections implement the **IDBElementListener** interface to manage refresh operations due to events.

5.10 Package ACTIONS

The **actions** package contains a couple of action classes. Each of these classes represents an action that is placed in one of the various context menus or associated with a button in the user interface. All action classes subclass the JFace **Action** class which in turn implements the **IAction** interface. The **IAction** interface offers methods to set the action label, the image associated with the action if it is used in a context menu and most importantly the **run** method which actually executes the action. The JFace **Action** class is a base implementation of the **IAction** interface. Subclasses usually override the **run** method.

5.11 Package WIZARDS

Wizards are special dialogs used when a modal operation requires a particular sequence for its information collection. Wizards have a title area along the top, a content area in the middle showing the various wizard pages and a button container at the bottom providing ‘Next’, ‘Back, ‘Finish’ and ‘Cancel’ buttons.
In the OMS front-end wizards are introduced to create the following OMS objects:

- Database using the **NewDatabaseWizard** class
- Object Type using the **NewTypeWizard** class
- Collection using the **NewCollectionWizard** class
- Association using the **NewAssociationWizard** class
- Method using the **NewMethodWizard** class

An additional wizard is used to import already existing OMS databases into the Eclipse workspace.
All wizard classes subclass the **Wizard** class which implements the **IWizard** interface and provides much of the **IWizard** behaviour. The **IWizard** interface contains methods to add the wizard pages and to get the current, the previous and the next page in the wizard. The task of the wizard classes in the OMS front-end is to create and initialise the pages it contains and execute the operation when the ‘Finish’ button is pressed.
In addition to subclassing the **Wizard** class, all wizards in the OMS front-end that create new objects and that are intended to be placed in the ‘New’ menu of the Eclipse workbench menu, are forced to implement the **INewWizard** interface. Moreover they are required to contribute to the **org.eclipse.ui.newWizards** extension point in the plug-in manifest. Implementing this extension point provokes Eclipse to automatically provide an action delegate for the wizard class and display it in the ‘New’ menu in the tool bar of the Eclipse workbench window. Further the OMS perspective adds shortcuts for these wizards in order to make them easily and fast accessible in the ‘New’ menu of the OMS perspective. The import wizard (class **ImportDatabaseWizard**) is added to the ‘Import’ menu of the Eclipse workbench and is therefore forced to implement the **IImportWizard** interface and the **org.eclipse.ui.importWizards** extension point. Figure 5.8 shows how the OMS front-end wizards integrate into the Eclipse wizard framework using the wizard to create a new OMS database as an example.

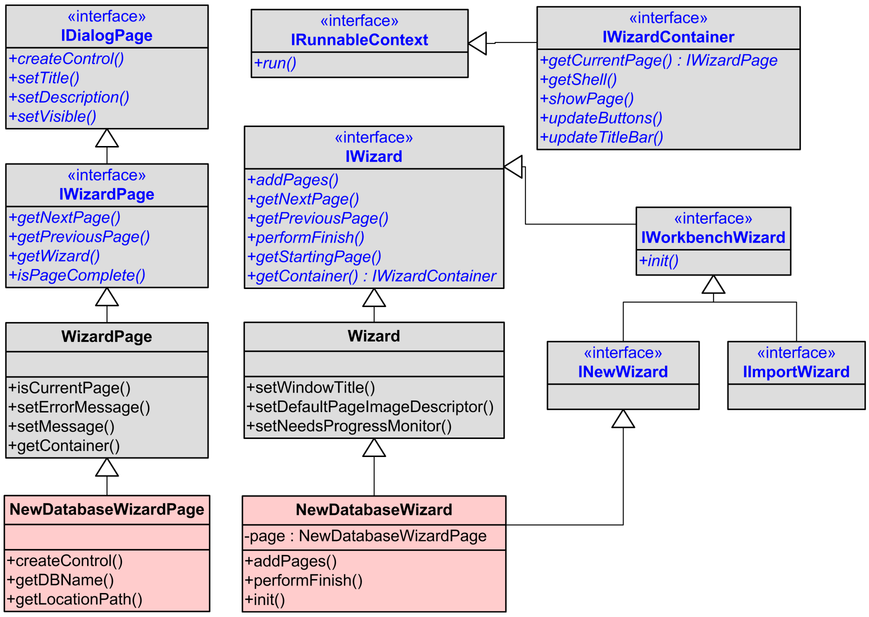

Figure 5.8: New OMS Database Wizard Classes

The actual wizard pages displaying the entries to fill in the requested data, extend the **WizardPage** class which implements the **IWizardPage** interface. The task of a wizard page is to present a page of information to the user, validate any information filled in by the user and provide accessors for the wizard class to gather the entered information.

5.12 Package PREFERENCES

The Eclipse platform offers a ‘preference framework’ which provides a mechanism for displaying options to the user and saving changed options across multiple Eclipse sessions. The OMS front-end uses this mechanism to add a specific OMS preference page to the Eclipse ‘Preferences’ dialog allowing the user to indicate the storage location of the OMS database system and an appropriate driver value.
The OMS preference page is hooked into the system by contributing an XML declaration to the **org.eclipse.ui.preferencePages** extension point. The actual logic to set up the page can be found in the **DBPreferencePage** class in the **preferences** package. The preference page is initialised with default values which are stored in the main plug-in class of the OMS front-end, namely the **OMSBrowserPlugin** class. If values are changed they are added to the preference store (interface **IPreferenceStore**) associated with the **OMSBrowserPlugin** class as well.

5.13 Package UTIL

The **util** package contains a conglomeration of classes which are used in the OMS plug-in.
The **DBObjectTransfer** and the **DBObjectDropTargetAdapter** are required in connection with drag and drop actions. The **DBObjectTransfer** class inherits from the **java.io.ByteArrayTransferClass** and is responsible for transferring database objects. A database object is identified by its object id and the parent object. Therefore, the **DBObjectTransfer** class writes these two values into a **java.io.DataOutputStream** and reads it again to restore the transferred object. The **DBObjectDropTargetAdapter** is a base implementation of the Eclipse **DropTargetListener** interface and is used to perform the actual drop operation.
Another group of Java classes in the **util** package are base implementations of content and label providers. The **DefaultContentProvider** implements the **IStructuredContentProvider** interface which can be used for all JFace table viewers. The **ListLabelProvider** and the **TableLabelProvider** are implemented to serve as basic label providers for JFace list viewers and table viewers respectively.
The **CollectionListDialog**, the **ObjectListDialog** and the **TypeListDialog** class are dialog classes allowing the user to choose from a list of possible collections, types or other objects. They all subclass the **SelectionDialog** class offered in the **org.eclipse.ui.dialogs** package.

6 Conclusions

6.1 Achievement

The goal of this diploma thesis was to design and implement a graphical user interface for OMS which is built on top of **OMS**^{jp}. **OMS**^{jp} is a Java interface that allows to access different OMS platforms in a uniform manner. The resulting OMS front-end is implemented as an Eclipse plug-in and enables the user to access OMS databases directly through the Eclipse environment.

6.1.1 Intended Goals

In the early stages of this diploma thesis the existing OMS Pro graphical user interface was evaluated. The aim was to derive requirements for the OMS front-end for Eclipse. The following list summarises briefly the requirements that were established in section 2.4.

- Keep the commit / rollback storing paradigm
- Offer context information
- Offer graphical schema editors
- Extend the object panels used by OMS Pro
- Redesign the object arrangement in the object area
- Provide an intuitive tab ordering

- Reduce the use of objects ids to make the working with objects easier
- Enhance the process of adding associations
- Introduce an application state memory

6.1.2 Discussion

The first requirement was to keep the storing paradigm of OMS Pro. The OMS front-end adopts the commit / rollback paradigm suggested by the existing OMS Pro GUI. This is against the Eclipse conventions but in the case of editing database objects instead of text files it makes sense to immediately take over any changes made. The user does not have to perform a lot of save operations and unintentional modifications can be discarded using the rollback functionality.
Context information is important to keep track of data in a system. Therefore the OMS front-end offers context information as required. The information is displayed directly in the corresponding editor windows which makes the information fast accessible and manageable. Any information section that the user does not want to see can be collapsed in order to concentrate on the essential parts of the editor window. For collection members a section with links to related objects is displayed. This section helps to simplify the process of adding links between two objects. If the user has one object open in the editor area, he just has to choose an appropriate object from a list dialog which can then be linked. The list does not only display object identifiers but it offers an attribute value in addition to the type and the id of the object. This fact enables the user to find the intended object without having to open it beforehand. This is a major improvement compared to the present OMS Pro GUI.
Another enhancement is the introduction of the database explorer which replaces the collection and macro panel of the OMS Pro user interface. These panels are a good approach to start with but they offer only a limited portion of information. However, the database explorer displays the most frequently used database objects in a clear tree structure. The user is able to expand only those tree nodes he requires in order to focus on the current task. The database explorer moreover offers a context menu with common actions to choose from. It is possible to create any kind of object displayed in the tree using the context menu. In most of the cases the new objects are created with the help of wizards. Wizards may be considered as circumstantial but the wizards used in the OMS front-end initialise the entry fields according to the user's selection when the wizard is called and accelerate thus the creation process. For example if the user wants to create a new association he can select two collections in the tree, call the wizard and he is able to finish the process only by typing the name of the association.
A further requirement was to reduce to use of object ids and to make working with objects easier. The OMS front-end uses different approaches to fulfil this task. One approach is to display collection members always together with an attribute value regardless if the object is shown as tree object or as tab bar label. Initially the first attribute value is taken but if this value does not help to identify objects quickly, the user can choose a different display attribute for every collection. If the user chooses a different attribute value for a collection, affected members of this collection are notified throughout the whole Eclipse user interface to change their display attribute value correspondingly. This gives the user a powerful tool

to identify objects without having to open them. Another approach to make working with objects less challenging is the introduction of icons. Every type of database objects has its own icon which is consistently used in the workbench. The user is able to visually recognise different object types. This helps to distinguish different kinds of objects and the user benefits if he wants to get back to an object which is already open in the editor area.

The possibilities for redesigning the object arrangement and the layout of the object area were rather limited by the Eclipse platform and its conventions. Having overlapping windows would have been possible, but it was neither what we aimed for nor would the concept have fitted into the workbench. Therefore the object area was designed to display complete object windows and only one at a time. More detailed thoughts about the available options are presented in section 4.1.1. The result is satisfying since the user is able to focus on the currently active object without being disturbed by other object windows in his field of vision. Furthermore the editor windows present important context information which reduces the need to have multiple objects open at the same time. The problem of the inefficient tab bar ordering was solved implicitly since the Eclipse framework already offers a mechanism for managing the tab bar.

As required, the OMS front-end offers an application state memory in different application parts. If the workbench is shut down, the application checks if a database was still connected and the next time the user chooses to restart Eclipse, the connection is restored. Moreover, the display attributes that the user can choose for every collection in order to determine how the collection members are presented are preserved across multiple sessions. File system dialogs memorise which directory was last accessed and the dialog directly selects this directory the next time it is launched.

The one requirement that was not satisfied is the introduction of a schema editor as described in section 4.1.3. Only the basic updating and browsing functionality was implemented. The main reason for this was that the process of really getting to know the whole Eclipse framework took longer than expected. Most of the books and material that are available treat quite simple plug-in examples which are all file-based. The OMS plug-in moves away from this file-based paradigm and it uses newly introduced Eclipse features which needed a lot of effort to get things working. Fortunately Eclipse has a large open source community which is very useful to exchange experiences and programming solutions and which advances the whole Eclipse project. Another reason that the schema editor implementation had to be skipped was due to the fact that the underlying **OMS**^{jp} library was still incomplete and erroneous. This slowed the implementation process down. It was time-consuming to find out if a problem was caused by the OMS plug-in or by the underlying library and what kind of problem it was.

Apart from the missing schema editor the OMS front-end meets all requirements and it allows to reason that we achieve the intended results. In our opinion the OMS front-end is a valuable alternative to the existing OMS Pro GUI making the modelling process of OMS databases easier and more intuitive. The benefits will become even more noticeable if **OMS**^{jp} offers drivers for other OMS platforms than OMS Pro since it will enable the users to manage different OMS databases without having to get acquainted with a new graphical user interface. Furthermore the OMS front-end integrates well into the Eclipse workbench since the editor windows use the same graphical components as the plug-in manifest editor and most of the conventions were respected.

6.2 Future Work

The OMS front-end already provides a lot of basic functionality and useful features to support the OMS modelling process. However, there is still an ample scope to apply improvements. The following list presents the possible enhancements.

- Schema Editor
  The OMS front-end improves the process of adding relations between objects. But the most intuitive way to add an association between two collections for instance is still the way of drawing the association in a graphical schema editor. Therefore the introduction of such an editor is a useful extension. Since Eclipse offers an elaborate plug-in architecture it is even possible to design an independent plug-in for the schema editor which extends the OMS plug-in.

- Drag and Drop
  Drag and drop is a useful tool to easily make connections between objects. The OMS front-end offers a basic drag and drop mechanism allowing the user to add collections members to a collection by dragging objects from the database explorer tree into the collection editor. The OMS front-end could be enhanced by offering more drag and drop operations that simplify other modelling tasks. For example to establish links between objects, it would be helpful if the user could drop an appropriate object directly on the link table in the 'Links' section. Another idea is to allow the user to drop an object on any other object to link these two objects if possible. If it is clear which association suits, the link would be added automatically, otherwise the system could show a dialog window to let the user choose from possible associations between these two objects.

- Icon View
  In the final front-end version the collection editor shows a list of collection members with one of their attribute values. At the beginning of this thesis one idea was to let the user choose how he would like to have the collection members displayed. The choices would be analogue to the Windows explorer where files can be viewed as detailed list or as large icons for example. Therefore the OMS plug-in could offer a second page in the collection editor showing all collection members as thumbnails providing more attribute values at first sight. It is even conceivable to display an image file as thumbnail if the object contains one.

- DDL Pages
  DDL, the Data Definition Language, is a language especially defined for OMS to specify an OMS schema definition. The DDL language can be used to refine a graphically developed data model or to define a data model from scratch. Expert users often know these definition languages very well and they would be much faster without the graphical support. Therefore an idea would be to add a page to all editor windows displaying the DDL sources of the according object. The user could then choose to modify an object directly by modifying the DDL source or using the graphical support.

- Complete Functionality
  The OMS front-end does not offer the whole range of functionality that OMS Pro does. For example it is not possible to check databases for consistency. If a database is inconsistent, there is no way to find out which constraints are violated. The missing functionality should be added to the front-end in order to support the complete modelling process.

A
Plugin XML Manifest

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>

<plugin
   id="ch.ethz.globis.omsjp.omsbrowser"
   name="OMSBrowser Plug-in"
   version="1.0.0"
   provider-name="ETH Zurich, Global Information Systems Group"
   class="ch.ethz.globis.omsjp.omsbrowser.OMSBrowserPlugin">

   <runtime>
      <library name="lib/jasper.jar">
         <export name="*"/>
      </library>
      <library name="lib/omspro.jar">
         <export name="*"/>
      </library>
      <library name="lib/omsjp.jar">
         <export name="*"/>
      </library>
      <library name="OMSBrowser.jar">
         <export name="*"/>
      </library>
   </runtime>

   <requires>
      <import plugin="org.eclipse.ui"/>
      <import plugin="org.eclipse.core.runtime"/>
      <import plugin="org.eclipse.jface.text"/>
```

```
    <import plugin="org.eclipse.core.resources"/>
    <import plugin="org.eclipse.ui.editors"/>
    <import plugin="org.eclipse.ui.ide"/>
    <import plugin="org.eclipse.jdt.core"/>
    <import plugin="org.eclipse.ui.forms"/>
    <import plugin="org.eclipse.ui.console"/>
</requires>

<extension
      point="org.eclipse.ui.editors">
    <editor
          class="ch.ethz.globis.omsjp.omsbrowser.editors.
              collection.CollectionEditor"
          icon="icons/unary_collection.gif"
          default="false"
          name="Collection Editor"
          id="ch.ethz.globis.omsjp.omsbrowser.editors.collection.
              CollectionEditor"/>
    <editor
          class="ch.ethz.globis.omsjp.omsbrowser.editors.object.
              ObjectEditor"
          icon="icons/single_object.gif"
          default="false"
          name="Object Editor"
          id="ch.ethz.globis.omsjp.omsbrowser.editors.object.
              ObjectEditor"/>
</extension>

<extension
      point="org.eclipse.ui.newWizards">
    <category
          name="OMS"
          id="ch.ethz.globis.omsjp.omsbrowser"/>
    <wizard
          finalPerspective="ch.ethz.globis.omsjp.omsbrowser.
              perspective.OMSPerspective"
          class="ch.ethz.globis.omsjp.omsbrowser.wizards.
              NewDatabaseWizard"
          icon="icons/database_connected_new.gif"
          category="ch.ethz.globis.omsjp.omsbrowser"
          name="OMS Database"
          id="ch.ethz.globis.omsjp.omsbrowser.wizards.
              NewDatabase"/>
    <wizard
          icon="icons/object_type_new.gif"
          class="ch.ethz.globis.omsjp.omsbrowser.wizards.
              NewTypeWizard"
          category="ch.ethz.globis.omsjp.omsbrowser"
          name="OMS Object Type"
          id="ch.ethz.globis.omsjp.omsbrowser.wizards.
```

```
                NewType"/>
      <wizard
            icon="icons/unary_collection_new.gif"
            class="ch.ethz.globis.omsjp.omsbrowser.wizards.
                NewCollectionWizard"
            category="ch.ethz.globis.omsjp.omsbrowser"
            name="OMS Collection"
            id="ch.ethz.globis.omsjp.omsbrowser.wizards.
                NewCollection"/>
      <wizard
            icon="icons/binary_collection_new.gif"
            class="ch.ethz.globis.omsjp.omsbrowser.wizards.
                NewAssociationWizard"
            category="ch.ethz.globis.omsjp.omsbrowser"
            name="OMS Association"
            id="ch.ethz.globis.omsjp.omsbrowser.wizards.
                NewAssociation"/>
      <wizard
            icon="icons/method_new.gif"
            class="ch.ethz.globis.omsjp.omsbrowser.wizards.
                NewMethodWizard"
            category="ch.ethz.globis.omsjp.omsbrowser"
            name="OMS Method"
            id="ch.ethz.globis.omsjp.omsbrowser.wizards.
                NewMethod"/>
   </extension>

   <extension
         point="org.eclipse.ui.perspectiveExtensions">
      <perspectiveExtension
            targetID="org.eclipse.ui.resourcePerspective">
         <perspectiveShortcut
               id="ch.ethz.globis.omsjp.omsbrowser.perspective.
                  OMSPerspective">
         </perspectiveShortcut>
      </perspectiveExtension>
      <perspectiveExtension targetID="org.eclipse.jdt.ui.
         JavaPerspective">
         <perspectiveShortcut id="ch.ethz.globis.omsjp.omsbrowser.
         perspective.OMSPerspective"/>
      </perspectiveExtension>
   </extension>

   <extension
         point="org.eclipse.ui.views">
      <category
            name="OMS"
            id="ch.ethz.globis.omsjp.omsbrowser">
      </category>
      <view
```

```
            name="Database Explorer"
            icon="icons/database_explorer.gif"
            category="ch.ethz.globis.omsjp.omsbrowser"
            class="ch.ethz.globis.omsjp.omsbrowser.views.
                DBExplorerView"
            id="ch.ethz.globis.omsjp.omsbrowser.views.
                DBExplorerView">
      </view>
      <view
            icon="icons/sample.gif"
            class="ch.ethz.globis.omsjp.omsbrowser.views.AQLView"
            category="ch.ethz.globis.omsjp.omsbrowser"
            name="AQL"
            id="ch.ethz.globis.omsjp.omsbrowser.views.AQLView"/>
      <view
            icon="icons/match_view.gif"
            class="ch.ethz.globis.omsjp.omsbrowser.views.MatchView"
            category="ch.ethz.globis.omsjp.omsbrowser"
            name="Match"
            id="ch.ethz.globis.omsjp.omsbrowser.views.MatchView"/>
   </extension>

   <extension
         point="org.eclipse.ui.perspectives">
      <perspective
            icon="icons/oms_logo.gif"
            class="ch.ethz.globis.omsjp.omsbrowser.perspective.
                OMSPerspectiveFactory"
            name="OMS"
            id="ch.ethz.globis.omsjp.omsbrowser.perspective.
                OMSPerspective"/>
   </extension>

   <extension
         point="org.eclipse.ui.importWizards">
      <wizard
            icon="icons/database_connected_new.gif"
            class="ch.ethz.globis.omsjp.omsbrowser.wizards.
                ImportDatabaseWizard"
            name="OMS Database"
            id="ch.ethz.globis.omsjp.omsbrowser.wizards.
                ImportDatabase"/>
   </extension>

   <extension
         point="org.eclipse.ui.preferencePages">
      <page
            class="ch.ethz.globis.omsjp.omsbrowser.preferences.
                DBPreferencePage"
            name="OMS"
```

```
            id="ch.ethz.globis.omsjp.omsbrowser.preferences.
                DBPreferencePage"/>
      <page
            class="ch.ethz.globis.omsjp.omsbrowser.preferences.
                AQLViewPreferencePage"
            category="ch.ethz.globis.omsjp.omsbrowser.preferences.
                DBPreferencePage"
            name="AQL View"
            id="ch.ethz.globis.omsjp.omsbrowser.preferences.
                AQLPreferencePage"/>
   </extension>

</plugin>
```

B User Manual

B.1 System Requirements

In order to use the OMS front-end, the following system components need to be installed on the computer:

- Windows operating system
- OMS Pro 3.0
- Eclipse 3.0

B.2 Installation

The OMS front-end can either be installed manually or with the help of the Update Manager provided by Eclipse.
To install the OMS front-end manually, please follow the instructions below:

1. Download the ‘OMSBrowser.zip’ file.
2. Unzip the contents of the ‘OMSBrowser.zip’ file into the ECLIPSE_HOME directory which is the main directory of the Eclipse installation (for example C:\Program Files\eclipse). The ECLIPSE_HOME\plugins directory should now contain two directories called ‘ch.ethz.globis.omsjp.omsbrowser_1.0.0’ and ‘ch.ethz.globis.omsjp.omsbrowser.help_1.0.0’.
3. If Eclipse was running during the unzip process, restart the Eclipse workbench.

For an installation with the help of the Update Manager, the following steps need to be accomplished:

1. Select the ‘Help → Software Updates → Find and Install’ menu entry to open the ‘Install/Update’ dialog.

2. Choose the option ‘Search for new features to install’ and advance to the next dialog window.

3. In order to download the OMS front-end a new ‘Remote Site’ has to be added with the URL of the OMS front-end update site. Mark the check box of the added update site to include it in the search process.

4. The search result presents the ‘OMS Browser Feature’ which can be selected. If the terms in the license agreement are accepted, the OMS plug-in is installed. Eclipse has to be restarted.

B.3 Getting Started

Setting OMS Properties

Before the OMS front-end can be used, it is necessary to set the general OMS properties comprising the OMS home directory and the **OMS**jp driver. In order to set these properties, open the OMS preference page (menu ‘Window → Preferences’) shown in figure B.1.

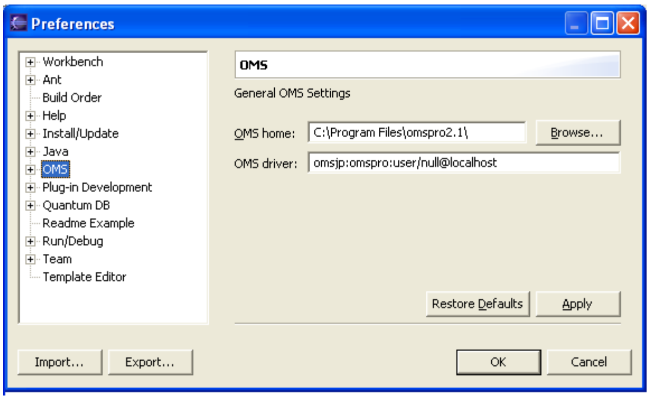

Figure B.1: OMS Preference Page

The form contains two fields to fill in. The first one is the home or installation directory of OMS Pro. The second entry contains the currently used **OMS**jp driver. The default value is ‘omsjp:omspro:user/null@localhost’ which represents the OMS Pro driver if OMS Pro is locally installed on the machine. If these values do not correspond to the user’s environment, the appropriate ones should be entered.

Opening the OMS Perspective

Having set the OMS properties, the OMS front-end is ready to use. To model OMS databases it is recommended to switch to the OMS perspective. To open the OMS perspective, choose

the ‘OMS’ perspective from the menu ‘Window → Open Perspective’. Initially the OMS perspective shows the database explorer view at the left and the console view at the bottom of the workbench window. The AQL view is hidden behind the console view.

Creating a New OMS Database

If the user wants to create a new OMS database, he can open the ‘New Database’ wizard shown in figure B.2. The wizard can be opened either by selecting the menu ‘File → New → OMS Database’ or by right clicking in the database explorer and choosing ‘New → OMS Database’.

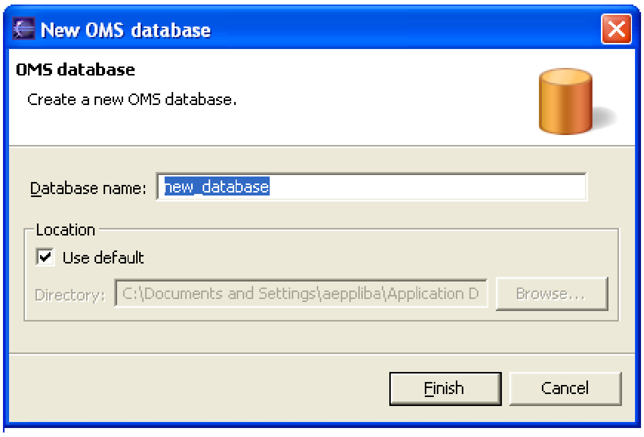

Figure B.2: New Database Wizard

To create a new database enter a name and a storage location if the database should not be stored in the default directory. By clicking ‘Finish’ the database is created and added to the database explorer. Since a database connection must be established for the creation process, the newly created database is already open in the database explorer and ready for browsing.

Importing an Existing OMS Database

The OMS front-end allows importing an already existing OMS database which is stored in the local file system. In order to import a database, open the ‘Import Database’ wizard. This can be done either by selecting the menu ‘File → Import’ and then choosing ‘OMS database’ or by right clicking in the database explorer and choosing ‘Import Database’. The wizard presented in figure B.3 is shown.
To import a database, fill in the location where the database is stored in the local file system and click ‘Finish’. The database is added to the database explorer.

Opening or Closing a Database

To open a database choose the ‘Open Database’ entry from the context menu in the database explorer. Please note that if another database is connected, the connection is closed since only

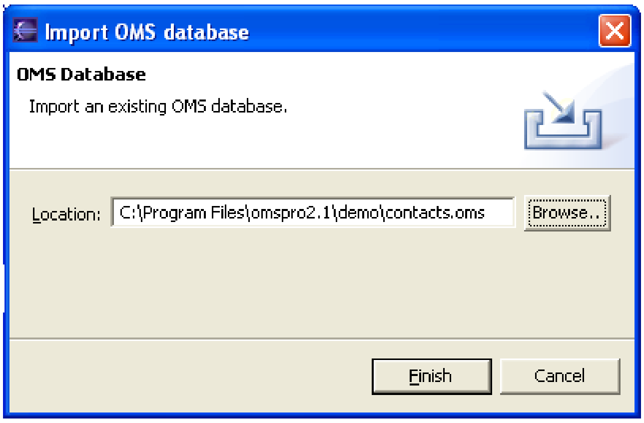

Figure B.3: Import Database Wizard

one active connection can be open at a time. To close an open database choose the ‘Close Database’ entry from the context menu.

B.4 Browsing and Updating the Database

The database explorer provides access to frequently used OMS database objects such as collections, types and macros. The system data is listed separately in the ‘System’ folder. Figure B.4 shows the database explorer and its context menu.

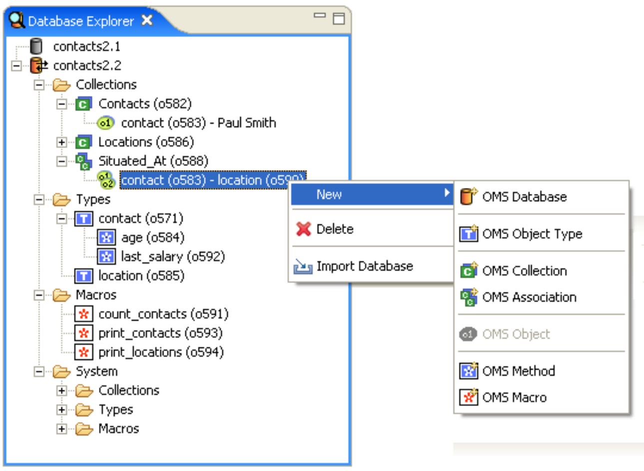

Figure B.4: Database Explorer

Browsing Collections

The ‘Collections’ folder contains first the unary collections followed by the binary collections. Expanding the collection nodes makes the collection extent visible.

Choosing a Display Value for a Collection

All collection members that belong to the collection extent of a unary collection are displayed by providing the type, the object id and one of their attribute values. Initially the first attribute value is taken for every collections. But the user can choose a different attribute if he likes. To do this access the context menu in the database explorer by right clicking on the appropriate collection and select the ‘Choose Attribute’ action. A dialog is opened where the user can choose the name of the attribute value that should be displayed for this collection. As a consequence of this, all members of this collection show the new attribute value throughout the whole Eclipse user interface.

Choosing the Sort Criteria in the Database Explorer

In the database explorer collections provide access to their members by displaying the object id and a single attribute value. The database explorer allows to determine the sort criteria using the local pull down menu of the database explorer as shown in figure B.5. The collection members can either be sorted by the object id or alphabetically according to the attribute value.

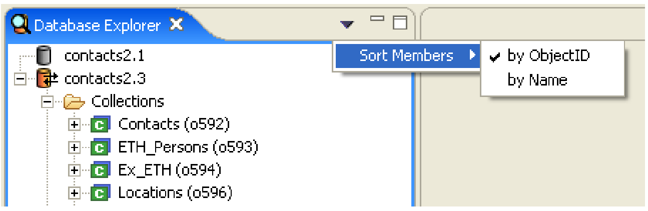

Figure B.5: Change Sort Criteria

Opening a Collection in the Editor Area

To open a collection in the editor area, double click on the corresponding collection in the database explorer. A collection editor window as shown in figure B.6 is opened in the editor area.

The editor window shows the collection extent as a table. The collection members in the table can be opened with a double click. If it is a binary collection either the source or the target object of a collection member can be opened depending on the table cell that is doubleclicked.

Creating a Unary Collection

A new unary collection can be created using the ‘Collection’ wizard. If the wizard is called using the ‘File → New’ menu then the wizard page is blank. If the context menu of the

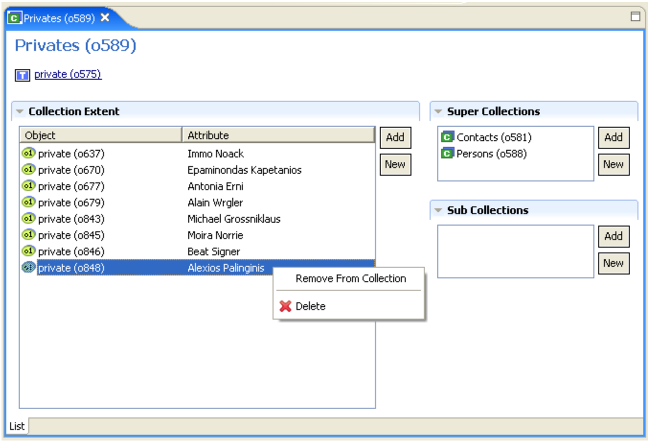

Figure B.6: Collection Editor

database explorer is used and launched by clicking on the ‘location’ type object, the collection type is already initialised accordingly as shown in figure B.7.

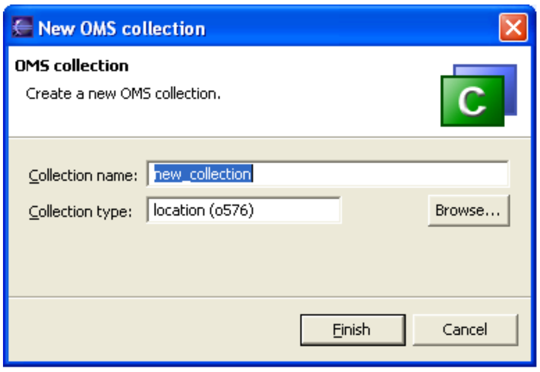

Figure B.7: Collection Wizard

The newly created collection is added to the database explorer and opened in the editor area.

Creating a Sub / Supercollection

In order to create a new sub- or supercollection, either the sub- and supercollection section in the collection editor or open the context menu on a collection in the database explorer can be used.

Creating a Binary Collection (Association)

Since no graphical schema editor is available, binary collections which are based on an association have to be created using a wizard as well. The fastest way to create an association is to mark the source and the target collection in the database explorer and call the ‘Association’ wizard as presented in figure B.8.

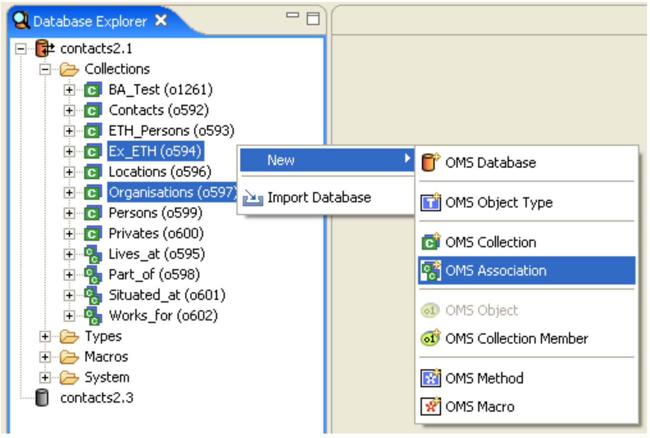

Figure B.8: Create a New Association

As a result the association wizard shown in figure B.9 is opened. The source and the target collection are initialised according to the selection in the database explorer. The 'Inverse' button can be used to swap the source and the target collection. The newly created association is added to the database explorer and opened in the editor area.

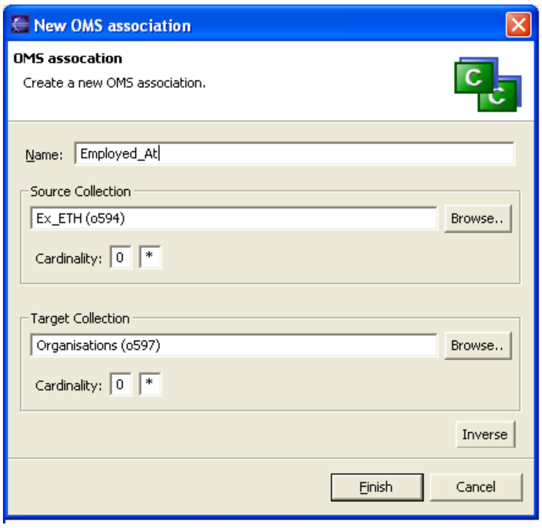

Figure B.9: Association Wizard

Browsing a Collection Member

Collection members are shown in the database explorer if the collection node is expanded. Members of unary collection can be opened in the editor area with a double click. The members are presented in object editor windows as shown in figure B.10.

The object editor has different areas and sections. The title area displays the attribute value

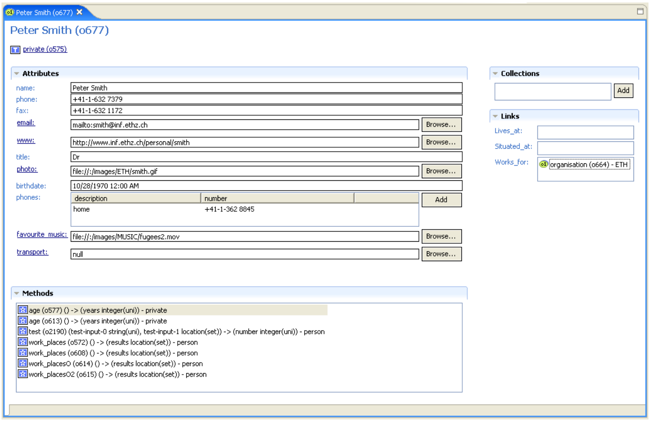

Figure B.10: Object Editor

chosen for the parent collection. Below a hyperlink to the type of the object is provided. The ‘Attributes’ section shows all attribute values of the object whereas all collections where this object is member, are listed in the ‘Collections’ section. The ‘Links’ section shows how this object is related to other database objects and the ‘Methods’ section presents all methods that can be executed on this object.

Creating a Collection Member

In order to create a new collection member either the ‘New’ button in the collection extent section of the collection editor can be pushed or the ’New → OMS Collection Member’ on a collection in the database explorer can be selected. This action does not need a wizard. The object is directly created and added to the collection where it was created. The newly created object is opened in the editor area.

Editing Attribute Values

The ‘Attributes’ section in the object editor allows editing of all attribute values of an object. Depending on the type of the attribute, the section offers a different form entry for displaying and editing the attribute value. If it is not clear what kind of type an attribute has, the user can move the mouse over the attribute label and a tool tip appears indicating the attribute type.
Simple text values such as for example string, integer or real values are displayed in a common text field which can be directly edited. Attributes of type ‘date’ offer an additional calendar to pick a date. If no time is indicated, the time value is set to ‘12:00 AM’ by default. Attributes of type ‘uri’ offer a button to browse the local file system. If the attribute represents a database object, the user can choose an appropriate object from a list dialog.
Structured attributes and bulk attribute values provide a context menu for the editing process by clicking the right mouse button on the attribute field. An additional dialog is opened to change or add new attribute values.

Accessing URIs

Attributes of type 'uri' can be accessed by clicking on the attribute label which is in this case a link . For 'http' entries the system's standard browser is opened whereas for 'mailto' entries the standard mail program is activated. Image files are shown in an appropriate application if the file path is absolute.

Adding an Object to another Collection

There are two possibilities to add an object to another collection. The first possibility is to open the collection to which the object should be added in the collection editor and then click the 'Add' button. A list dialog presents all objects that have the same type as the target collection and that are not yet member of the collection. The dialog is shown in figure B.11. The objects that are selected are subsequently added to the collection.

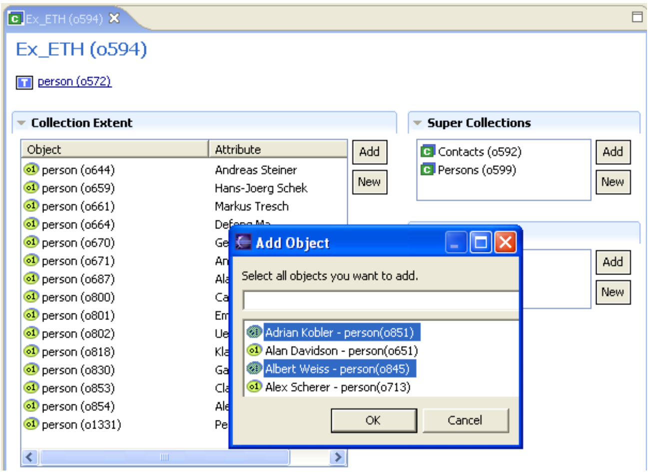

Figure B.11: Adding Objects to a Collection

The second possibility is to use drag and drop by dragging objects from the database explorer into the collection extent table in the collection editor. In this case it is also possible to add members of a sub- or supertype of the collection membertype. The drag and drop process is shown in figure B.12.

Adding a Link between Two Objects

The easiest way to establish a link between two objects is to open one of these objects in the object editor and to make use of the 'Links' section. Every link table in the 'Links' section offers a context menu which allows to add a new link. A list dialog provides all objects of the target collection to choose from.
Another possibility is to use drag and drop. If a binary collection is open in the collection editor, one object that is member of the source collection and one object that is member of the target collection can be selected in the database explorer. Dropping these objects on the collection extent of the binary collection, adds a new binary collection member. If the source

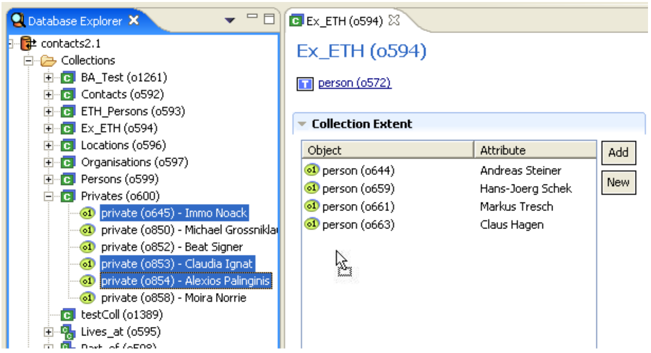

Figure B.12: Adding Objects to a Collection using Drag&Drop

and the target collection are identical, a dialog appears to let the user decide which object to take as source object. Figure B.13 visualises the process.

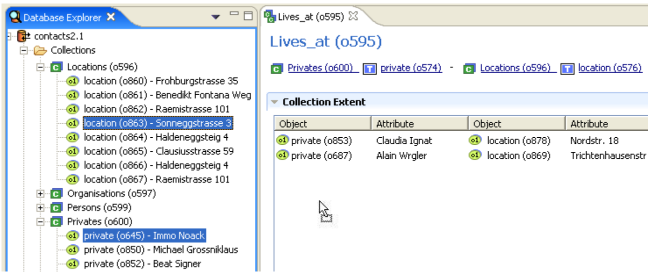

Figure B.13: Adding Objects to a Binary Collection using Drag&Drop

Removing Collection Members

Collection members can be removed from a collection either using the corresponding action in the context menu of the collection editor or in the database explorer. It is possible to remove more than one object at the same time using the collection editor.

Browsing Types

The database explorer provides access to user and system types. To open a type object it can be double clicked and an object editor window shows up in the editor area. The editor window provides the common ‘Attributes’ section, a ‘Supertype’ and a ‘Subtype’ section to view, add or remove sub- and supertypes and a ‘Method’ section to add and remove new methods.

Creating Types

New object types are created using a the ‘Type’ Wizard shown in figure B.14.

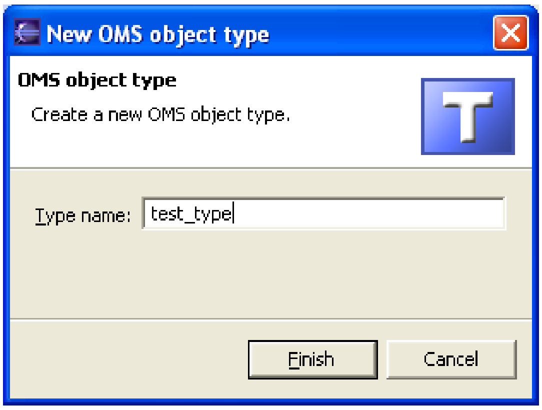

Figure B.14: Object Type Wizard

To create instances of base types or structured types, no special wizard is available. The user has to use the 'New → OMS Object' action in the database explorer on the 'btype' and 'stype' object respectively.

Browsing and Creating Macros

All user and system macros are listed in the database explorer. They can be opened in an object editor window providing the 'Attributes' section. In order to execute macros, the 'Attributes' section offers a 'Run' button. Unfortunately the OMS front-end is not able to show any side effects of the macro. It only indicates if the execution was successful or not.
To create a new macro the user can choose the 'New → OMS Macro' action in the database explorer. The newly created macro is directly opened in the editor area.

Browsing and Creating Methods

Methods are associated with object types, therefore they are listed in the database explorer as child objects of the type object they belong to. New methods are created with the method wizard presented in figure B.15.

Creating Objects

For the most frequently used database objects, special wizards are provided to create new instances of these objects. All other kinds of objects can be created by using the 'New → OMS Object' action which is provided in the context menu of the database explorer. The action is enabled if it is called on an object type in the resource tree. As a result an object based on the selected object type is created and opened in the editor area.

Retrieving Objects

Objects which are not listed in the database explorer tree can be retrieved by using the match functionality possibly known from OMS Pro. On every object type a 'Match' operation can

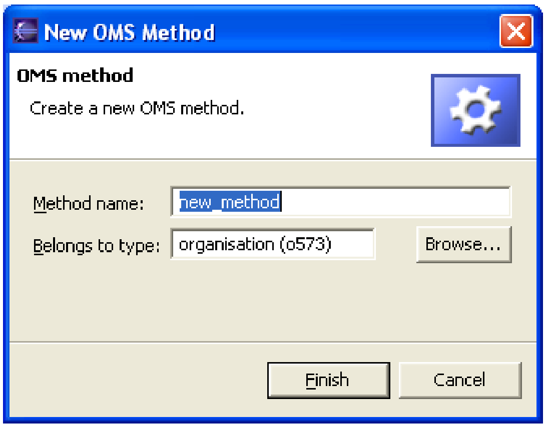

Figure B.15: Method Wizard

be executed. All objects that are based on this object type are displayed in a view especially opened for such a request. The view can be compared to the 'Search' view in the 'Java' perspective of Eclipse. Two different match result windows are presented in figure B.16. By double clicking the objects listed in the match view table, they are opened in the editor area and it is possible to edit their attributes. Depending on the kind of object, attribute editing operations need to be executed carefully since the database might not be consistent anymore after the editing process. For example if the user changes the source collection of an association, the members of the associated binary collection might have the wrong format and in this case it is no longer possible to open these objects.

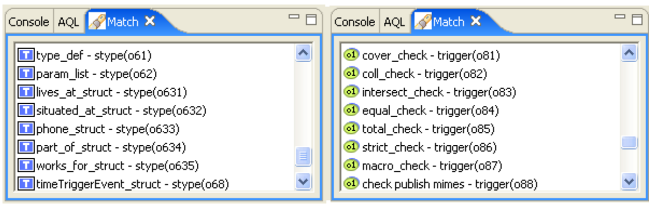

Figure B.16: Two Match Result Windows

Deleting Objects

Any kind of database objects can be deleted using the context menu in the database explorer or in some of the editor windows. If the user executes the 'Delete' action, the selected object is deleted in the OMS database and it is removed everywhere in the OMS perspective.
To delete a database, a dialog is displayed which allows the user to choose if he just wants to remove the database from the Eclipse environment or if the database should be deleted entirely from the file system. For members of unary collections the user can choose between the 'Delete' and the 'Remove From Collection' action. Choosing 'Delete' provokes the deletion of the database object and additionally the database object is removed from all unary collections it has been member. References to the object in binary collections or attribute

values need to be removed manually. The ‘Remove From Collection’ action only removes the selected object from the corresponding collection without deleting it. Members of binary collections cannot be deleted, but only removed from the collection.
If the user executes a ‘Match’ operation, all matching objects are listed in the ‘Match’ view. The table in the match view offers a ‘Delete’ action as well, to allow the user to delete any kind of database objects. If this delete action is used, the database object is deleted, but there are no further delete operations to restore the database consistency and the OMS perspective is not notified. Refresh operations have to be done manually.

Committing a Database

The context menu of the database explorer offers a menu entry to commit the currently open database. The OMS console informs the user if the commit operation was executed successfully or not. If it failed, there is no possibility to find out why it failed.

Undoing Updates by Rollback

Any modifications since the last commit can be undone by choosing the ‘Rollback Database’ functionality in the context menu of the database explorer.

B.5 Querying the Database

The OMS front-end offers an AQL view which allows to query the database. In the OMS perspective the AQL view is by default placed behind the console window. It can be activated using the tab in the tab bar or if it is closed by choosing the ‘Window → Show View → AQL’ menu entry. The AQL window is shown in figure B.17.

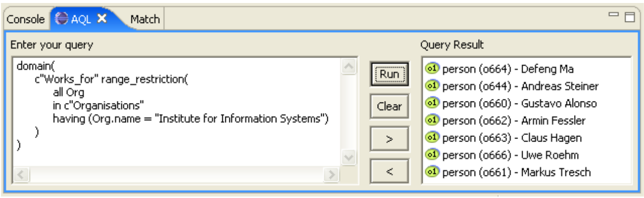

Figure B.17: AQL View

Executing a Query

To execute a query the query text has to be entered in the text field on the left side of the window. If more information is needed about how to write queries, please consult the OMS Introductory Tutorial [7]. The ‘Run’ button executes the query and the result is displayed in the table on the right side or the system informs the user that the execution of the query failed. If the result consists of database objects, they can be opened with a double click.

Setting the History Preference

The AQL view provides a history functionality which means that executed queries are stored. These query statements can be retrieved using the arrow buttons to go back and forth in the query list.

The AQL view has its own preference page (‘Window → Preferences → OMS → AQL’) which allows indicating how many queries should be available in the history list.

Acknowledgements

First of all, I would like to thank my supervisor Michael Grossniklaus for his valuable feedback, for reviewing my report and for the great effort he put into **OMS**^{jp} to make my front-end work. Furthermore, I want to thank Prof. Moira C. Norrie for giving me the opportunity to accomplish my diploma thesis in her group. I also would like to thank Ljiljana Vukelja for her constructive comments on the user interface design and Olivier Jeger for helping me with the design of the icons. Thanks to Rolf Bruderer, Olivier Jeger and especially Jean-Daniel Merkli for kindly supporting me during this thesis. Finally, I am grateful to my parents for their support in every respect.

Bibliography

[1] E. Clayberg and D. Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Addison-Wesley, 2004.

[2] Eclipse.
`http://www.eclipse.org`.

[3] N. Edgar, K. Haaland, J. Li, and K. Peter. Eclipse User Interface Guidelines. February 2004.
`http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html`.

[4] M. Grossniklaus. OMS^{jp}: A Uniform Java Interface to Heterogenous OMS Platforms. Technical White Paper, April 2004. Version 1.0.

[5] B. Majewski. A Shape Diagram Editor. December 2004.
`http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html`.

[6] M.C. Norrie. An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems. In *Proc. 12th Int. Conf. on Entity-Relationship Approach*. Springer-Verlag LNCS 823, 1993.

[7] M.C. Norrie, A. Würgler, K. von Gunten A. Palinginis, and M. Grossniklaus. OMS Pro 2.0: Introductory Tutorial. Technical Report, March 2003.

[8] OMS - Object Model System.
`http://www.oms.ethz.ch`.

[9] OMS^{jp}.
`http://www.oms.ethz.ch/omsjp`.

[10] SICStus Prolog.
`http://www.sics.se/sicstus/`.

[11] D. Springgay. Using Perspectives in the Eclipse UI. *IBM OTI Labs*, August 2001.
`http://www.eclipse.org/articles/using-perspectives/PerspectiveArticle.html`.

[12] TCL - Tool Command Language / Tk - User Interface Toolkit.
`http://www.tcl.tk/software/tcltk/`.

[13] S. Tilkov. Eclipse Forms Programming Guide. February 2004.

[14] P. Zoio. Building a Database Schema Diagram Editor with GEF. September 2004. http://www.eclipse.org/articles/Article-GEF-editor/gef-schema-editor.html.